Algoritmia básica

Algoritmia básica

Roberto Flórez Rueda

Rector de la Universidad de Antioquia
Alberto Uribe Correa

Vicerrector de Docencia
Óscar Sierra Rodríguez

Decano de la Facultad de Ingeniería
Carlos Alberto Palacio Tobón

Vicedecano de la Facultad de Ingeniería
Julio César Saldarriaga Molina

Asesor metodológico del Programa de Educación Ude@
Guillermo León Ospina Gómez

Autor
Roberto Flórez Rueda

Jefe del Departamento de Recursos de Apoyo e Informática (DRAI)
Juan Diego Vélez Serna

Coordinadora de Producción
Lyda Yaneth Contreras Olivares

Corrector de estilo
Daniel Aldana Estrada

Diagramación y diseño
Juan Felipe Vargas Martínez
César Augusto Pineda Duque

Impresión
Imprenta Universidad de Antioquia

Segunda edición, julio de 2011

Esta publicación es un producto del Programa de Educación Virtual Ude@. Reservados todos los derechos. No se permite la reproducción, archivo o transmisión total o parcial de este texto mediante ningún medio, ya sea electrónico, mecánico, óptico, de fotorreproducción, memoria o cualquier otro tipo sin permiso de los editores Ude@.

© Universidad de Antioquia
ISBN: 978-958-714-439-0

Impreso en Medellín (Colombia)

Dedicatoria

A mis padres, mis hermanos, mi esposa y mi hijo
porque me quieren mucho

Agradezco a Dios
porque me lo ha dado todo

Acerca del autor

Roberto Flórez Rueda

Roberto Flórez Rueda, Ingeniero Civil (1976) de la Universidad Nacional (sede Medellín), es profesor titular de la Universidad de Antioquia. Trabajó como analista programador en Tejicóndor y en Molinos Nutibara entre 1977 y 1978, y fue jefe de sistemas en Textiles Modernos desde 1979 hasta 1982, fecha a partir de la cual se vinculó como docente en el Departamento de Ingeniería de Sistemas de la Universidad de Antioquia. Ha participado en los proyectos "Estudio de actualización de demanda", para el Metro de Medellín; "Estudio de accidentalidad en las troncales de Antioquia y Córdoba", para el INTRA; "Base de datos jurídica", para la División Jurídica del municipio de Medellín; "Sistema de control del ordenamiento físico", para Planeación Metropolitana del municipio de Medellín; y "Sistema integrado para caracterización, recolección y transporte de desechos sólidos", para Empresas Varias del municipio de Medellín. Fue jefe del Departamento de Ingeniería de Sistemas de la Universidad de Antioquia entre 1987 y 1989. En la actualidad imparte en esta Institución los cursos *Algoritmos, Estructuras de datos* y *Teoría de compiladores*, y es profesor de cátedra en la Escuela de Ingeniería de Antioquia, donde dicta los cursos *Estructuras de datos* y *Fundamentos de programación y compiladores*.

Correo electrónico: rflorez@jaibana.udea.edu.co

Cómo usar este texto

Como estudiante del Programa de Educación Virtual de la Universidad de Antioquia, Ude@, usted es el centro del modelo educativo y puede controlar el proceso de aprendizaje mediante la organización del tiempo alrededor de sus intereses. La autonomía, la disciplina, la creatividad y el trabajo en equipo son características que le ayudarán en su formación para solucionar problemas reales de la sociedad, recurriendo al método de la ingeniería.

Los cursos Ude@ permiten fortalecer estas características mediante el desarrollo de diferentes actividades:

- Estudio individual, apoyado en diferentes medios (impresos, audiovisuales, multimedia).
- Estudio en grupo y acompañamiento del profesor a través del aula virtual.
- Tutorías presenciales, cuya finalidad es apoyar el aprendizaje y afianzar los temas estudiados.

El texto Ude@

En el modelo Ude@ los contenidos educativos son aportados por cada medio teniendo en cuenta las fortalezas propias de cada uno de ellos. Desde el punto de vista pedagógico, el texto impreso es por tradición un medio idóneo para los procesos de educativos ya que facilita el aprendizaje de hechos, la comprensión de principios generalizados o abstractos y el desarrollo del razonamiento lógico. En estos aspectos, el texto Ude@ es un medio muy eficaz para desarrollar y adquirir tales destrezas.

Estructura del texto

El texto *Algoritmia básica* ha sido desarrollado como parte del material educativo de los estudiantes del programa; sin embargo, su contenido puede ser de gran utilidad para cualquier persona que desee estudiar este tema.

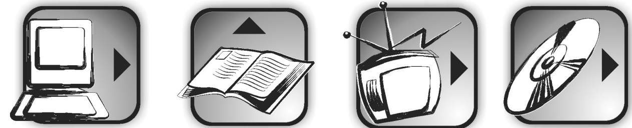
La estructura del texto es lineal, con una progresión gradual de cada tema, lo cual hace más fácil la transmisión del contenido de una manera lógica.

La división del texto está dada por módulos. Cada módulo contiene, en su primera página, una introducción, los objetivos de aprendizaje, unas preguntas básicas (relacionadas con los conocimientos previos requeridos) y el índice temático del contenido, que le guiarán en el proceso de aprendizaje sobre el tema en particular de cada sesión de clase.

Los iconos y la interrelación de medios

El material Ude@ ha sido producido de manera integral, teniendo como objetivo primordial el autoaprendizaje. Por tanto, la producción de los contenidos se desarrolla en los diferentes formatos (audiovisuales, web, multimedia, videoconferencias), con enlaces entre los mismos. La esencia de estos enlaces está dada por los iconos Ude@.

Los iconos, como representaciones gráficas de la realidad, serán los elementos gráficos que le ayudarán a guiarse en su navegación por los diferentes medios.



Sugerencias para los estudiantes

En la lectura del libro:

- Antes de iniciar el estudio de un capítulo, lea el contenido breve y la presentación.
- Trate de resolver las preguntas básicas de cada módulo; estas preguntas están diseñadas para ayudarle a comprender los conceptos o temas presentados a lo largo del mismo.
- Lea los ejemplos intercalados en los bloques de texto y trate de resolver los ejercicios con el fin de mejorar sus habilidades en la solución de problemas reales.
- Complemente la lectura del libro con las herramientas de comunicación que posee en el aula virtual y en su correo electrónico.
- Recuerde que sobre el tema que está estudiando en el módulo impreso también existe material disponible en otros medios, y que ese material representa valor agregado puesto que el contenido de los diferentes formatos no se repite sino que se complementa.

En el aula virtual:

- Aprenda cómo funcionan las herramientas indispensables para participar en un curso por red: sistema de correo electrónico, sistema de chat, grupos de discusión, búsquedas en Internet, consulta en bases de datos especializadas, entre otras.
- Revise el correo electrónico todos los días.
- Visite con relativa frecuencia el sitio Ude@ y la plataforma donde se publica el curso en Internet para enterarse de cualquier nueva información. Apóyese en la red como un sistema de consulta y establezca criterios para seleccionar la información requerida.
- Introduzca sus datos personales en el aula virtual para que sus tutores y compañeros tengan acceso a ellos.
- Desarrolle, en la primera semana, las actividades preparativas para el curso indicadas en el aula virtual.
- Dedique al menos tres horas semanales por cada crédito asignado al curso para leer los módulos, realizar trabajos, participar en los foros de discusión y presentar evaluaciones, de acuerdo con lo establecido en el cronograma.
- Planee su agenda personal para participar activamente en cada curso y entregar oportunamente sus tareas. En caso de algún imprevisto, debe comunicarse inmediatamente con el tutor.
- Participe de las actividades propuestas para realizar en forma individual y en grupos de trabajo. Haga parte de grupos de trabajo conformados con sus compañeros de curso y en ningún caso pretenda realizar todas las actividades sin ayuda de los demás.
- Manifieste oportunamente a sus compañeros y al profesor las dificultades que se le presentan con las actividades propuestas.
- Elabore su propio horario de trabajo independiente para el curso y cumpla con el cronograma propuesto.
- Realice con honradez las actividades de evaluación, autoevaluación y coevaluación que encuentre programadas en el curso.
- Durante su proceso de aprendizaje trate de adquirir autonomía con el conocimiento, es decir, intente construir nuevos conocimientos recurriendo a fuentes de información bibliográfica y a sus habilidades de comparación, análisis, síntesis y experimentación.
- Mantenga una actitud de colaboración con compañeros, tutores y monitores, y esté siempre dispuesto a realizar las actividades de aprendizaje.
- Relaciónese de manera respetuosa y cordial con los demás estudiantes, con el tutor y con los monitores.

Tabla de contenido

Capítulo 1
Herramientas a utilizar
Pág. 21

Módulo 1
Panorama general 23

Módulo 2
Elaboración de algoritmos y representación de datos 27

Capítulo 2:
Estructuras básicas en el desarrollo de algoritmos
Pág. 33

Módulo 3
Estructuras para la construcción de algoritmos e instrucciones de lectura y escritura 35

Módulo 4
Instrucción de asignación y expresiones aritméticas 39

Módulo 5
Manipulación de expresiones algebraicas 45

Módulo 6
Expresiones relacionales y lógicas 49

Módulo 7
Aplicaciones del capítulo 53

Capítulo 3:
Estructura decisión
Pág. 59

Módulo 8
Estructura decisión 61

Módulo 9
Componente DE_LO_CONTRARIO 65

Módulo 10
Decisiones anidadas 69

Módulo 11
Selección múltiple 75

Módulo 12
Ejemplo de uso de la instrucción CASOS 81

Capítulo 4:
Estructura ciclo
Pág. 85

Módulo 13
Estructura ciclo e instrucción MIENTRAS 87

Módulo 14
Contadores y acumuladores 93

Módulo 15
Aplicaciones de ciclos en matemáticas 99

Módulo 16
Ciclos anidados con instrucción MIENTRAS 105

Módulo 17
Instrucción PARA 111

Módulo 18
Ciclos anidados con instrucción PARA 117

Módulo 19
Instrucción HAGA 121

Capítulo 5:
Subprogramas
Pág. 125

Módulo 20
Concepto e identificación 127

Módulo 21
Clases y aplicación 133

Módulo 22
Parámetros y variables 137

Módulo 23
Ejemplos de uso 143

Módulo 24
Ejemplos de uso mejorados 147

Capítulo 6:
Vectores
Pág. 151

Módulo 25
Identificar el uso de un vector 153

Módulo 26
Definición y construcción 159

Módulo 27
Operaciones básicas: mayor dato, menor dato e intercambio de datos 163

Módulo 28
Operaciones básicas: proceso de inserción en un vector ordenado ascendentemente 169

Módulo 29
Operaciones básicas: proceso de borrado en un vector 173

Módulo 30
Operaciones básicas: búsqueda binaria y ordenamiento por selección 177

Módulo 31
Operaciones básicas: ordenamiento por burbuja 187

Capítulo 7:
Estructura estática de dos dimensiones: matriz
Pág. 195

Módulo 32
Definición, identificación y construcción de una matríz 197

Módulo 33
Recorridos sobre matrices 201

Módulo 34
Suma de los datos de filas y columnas de una matriz 207

Módulo 35
Clases de matrices 213

Módulo 36
Intercambio de filas y columnas de una matriz 219

Módulo 37
Ordenamiento de los datos de una matriz con base en los datos de una columna 223

Módulo 38
Construcción de la transpuesta de una matriz y suma de matrices 227

Módulo 39
Multiplicación de matrices 231

Capítulo 8:
Programación orientada a objetos
Pág. 235

Módulo 40
Introducción a la programación orientada a objetos 237

Módulo 41
Desarrollo de los métodos de la clase vector 247

Módulo 42
Herencia y polimorfismo dinámico en vectores 255

Módulo 43
La clase matriz 265

Bibliografía y cibergrafía
Pág. 271

Prólogo

En este libro se presenta una introducción detallada al arte de elaborar algoritmos y se muestran y explican las estructuras básicas para construirlos. Dichas estructuras, cada una de ellas conformada por un grupo de instrucciones, son la estructura secuencia, la estructura decisión y la estructura ciclo. Para cada una se exponen sus características especiales, sus ventajas y desventajas y los casos en los cuales se usan.

El libro hace énfasis en la programación modular, que es una vía para llegar a la programación orientada a objetos.

El autor

Objetivo general

Desarrollar la habilidad para elaborar algoritmos correspondientes a soluciones que serán implementadas como programas de computador.

Objetivos específicos

1. Ubicar la elaboración de algoritmos dentro del desarrollo de soluciones utilizando el computador como herramienta.

2. Reconocer la estructura secuencia con sus instrucciones de lectura, escritura, asignación, decisión y ciclo.

3. Reconocer la estructura decisión y utilizar sus instrucciones: SI, su componente opcional DE_LO_CONTRARIO, y CASOS.

4. Reconocer la estructura ciclo y utilizar sus instrucciones: MIENTRAS, PARA y HAGA.

5. Identificar las diferentes tareas que se ejecutan en un algoritmo para diseñar cada una de ellas como un subprograma y poder elaborar algoritmos más legibles, más sencillos y reutilizables.

6. Identificar y usar la estructura arreglo en sus formas básicas: vectores y matrices.

Capítulo 1 Herramientas a utilizar

Contenido breve

Módulo 1
Panorama general

Módulo 2
Elaboración de algoritmos y representación de datos

Para iniciar estudios referentes a lo que es la elaboración de algoritmos debemos conocer cuáles son las herramientas que se utilizan. Dichas herramientas incluyen el componente físico (*hardware*) y el componente lógico (*software*).

En el módulo 1 se presenta el esquema general de un computador, se describe brevemente la función de cada una de las partes que conforman la máquina, se describen los pasos que se siguen para desarrollar soluciones teniendo la máquina como herramienta y se ubica el curso dentro de ese desarrollo.

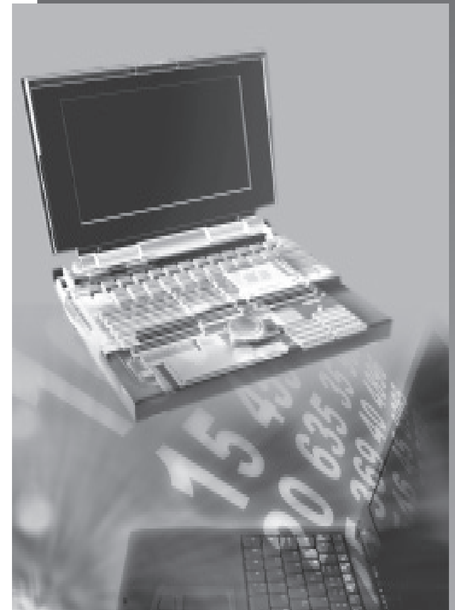
En el módulo 2 se describen los pasos que se siguen en la elaboración de algoritmos y los elementos con los cuales se construyen algoritmos: estructuras lógicas y datos.

Entorno de programación y utilización de computadores.

Módulo 1
Panorama general

Introducción
En este módulo se presenta el esquema general de un computador (herramienta sobre la cual trabajará el estudiante que se introduce al mundo de la informática) y los pasos que se siguen en el desarrollo de soluciones.

En la definición de los pasos que se siguen en el desarrollo de soluciones, utilizando el computador como herramienta, se ubica el curso que vamos a desarrollar.



«Conocer un computador y las partes físicas y lógicas que lo componen es de gran ayuda para comprender la revolución informática que nos ha absorbido en los últimos años y que ha afectado notoriamente el estilo de vida de la humanidad».

Objetivos del módulo
1. Presentar las herramientas básicas para construir soluciones utilizando un computador.
2. Ubicar el curso dentro del proceso de construcción de soluciones.

Preguntas básicas
1. ¿Cuáles son los componentes de un computador?
2. ¿Qué es la CPU?
3. ¿Cuáles son los pasos en el desarrollo de soluciones?
4. ¿Dónde se ubica el curso en el desarrollo de soluciones?

Contenidos del módulo
1.1 Esquema general de un computador
1.2 Desarrollo de aplicaciones

Vea en el botón **Algoritmia básica** del mapa conceptual el video "Módulo 1. Panorama general".

1.1 Esquema general de un computador

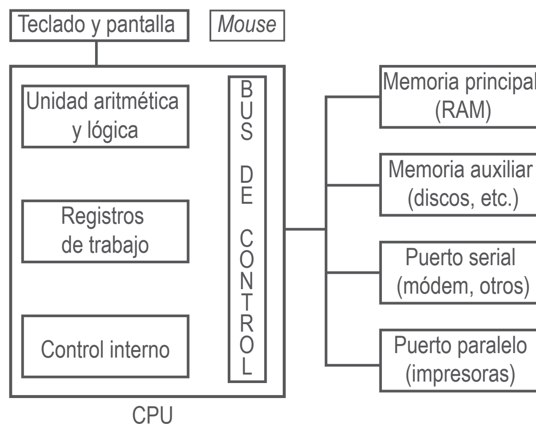


Figura 1.1. Componentes de un computador.

Expliquemos brevemente cada uno de los componentes de un computador (figura 1.1).

- La *CPU* (**C**entral **P**rocessing **U**nit) es el cerebro de la computadora y controla la operación de la totalidad del sistema de cómputo. Esta unidad inicia o realiza todas las referencias de almacenamiento, manipulación de datos y operaciones de entrada y salida.
- La *unidad aritmética y lógica* es la encargada de efectuar las operaciones de suma, resta, multiplicación, división y comparación.
- Los *registros de trabajo* son registros en donde se hallan los datos y las instrucciones sobre los cuales actúa la unidad aritmética y lógica en un momento dado.
- El *control interno* se encarga de determinar cuáles son las operaciones que se ejecutan, sobre cuáles datos y con cuáles instrucciones.
- El *bus de control* transporta señales de estado de las operaciones efectuadas por la CPU con las demás unidades, las cuales se denominan periféricos. La unidad central de procesamiento interactúa con los periféricos: teclado, pantalla, cintas, discos, etc.
- La *memoria principal* (RAM) es el sitio donde se hallan los datos listos para ser trasladados a los registros de trabajo de una manera eficiente.
- La *memoria auxiliar* es el conjunto de dispositivos externos en los cuales se almacena la información.
- Los *puertos seriales* conectan con módems, *mouse*, memorias electrónicas, etc.
- El *puerto paralelo* maneja la impresora.

1.2 Desarrollo de aplicaciones

Cuando se va a desarrollar una aplicación usando el computador como herramienta se tiene establecida cierta metodología para garantizar que la aplicación desarrollada sea de buena calidad. Los pasos que establece dicha metodología son:

1. Análisis del problema
2. Diseño de la solución
3. Implementación de la solución planteada

   3.1 Elaboración de algoritmos
      3.1.1 Análisis del problema
      3.1.2 Diseño de la solución
      3.1.3 Construcción del algoritmo
      3.1.4 Prueba de escritorio
   3.2 Codificación en algún lenguaje de programación
   3.3 Compilación
   3.4 Pruebas del algoritmo
4. Pruebas del sistema
5. Puesta en marcha

- El *análisis del problema* consiste en estudiar el problema al cual se le desea dar solución utilizando el computador como herramienta. Este estudio incluye entrevistas con los usuarios y con todas las personas que conocen cómo resolver el problema externamente al computador; se investigan los datos que se requieren, la información que se desea producir y los casos especiales que se puedan presentar. El análisis es fundamental. Un buen análisis tendrá como consecuencia un buen diseño.
- El *diseño de la solución* consiste en elaborar un modelo en el cual se consideran todas las situaciones que se identificaron en el análisis: se definen bases de datos, archivos, procesos, recursos y todo lo necesario para solucionar el problema.
- La *implementación* consiste en conseguir los recursos necesarios, construir las bases de datos y los archivos y elaborar todos los programas para que el sistema funcione apropiadamente.
- Las *pruebas* consisten en suministrar, al modelo desarrollado, datos ficticios con el fin de comprobar que el sistema responde correctamente a todas las situaciones que se hallaron en el análisis y que se implementaron como programa de computador. Esta fase de pruebas es muy importante puesto que allí se pueden detectar fallas y situaciones que no se consideraron, por alguna razón, en el análisis y en el diseño. Las pruebas deben estar supervisadas en forma exhaustiva por los usuarios del sistema y por sus desarrolladores.
- La *puesta en marcha* consiste en liberar el sistema desarrollado, para que el usuario lo utilice en su labor diaria. En esta fase de puesta en marcha se hace seguimiento al funcionamiento de la aplicación para efectuar los últimos ajustes en caso de que sea necesario.

Nuestro curso se centra en lo correspondiente al numeral 3.1.

Resumen

En este módulo hemos tratado lo correspondiente al esquema general de un computador y a la ubicación del curso en el desarrollo de soluciones utilizando el computador como herramienta.

Ejercicios propuestos

1. ¿Cuáles son los pasos que se siguen para elaborar soluciones utilizando el computador como herramienta?

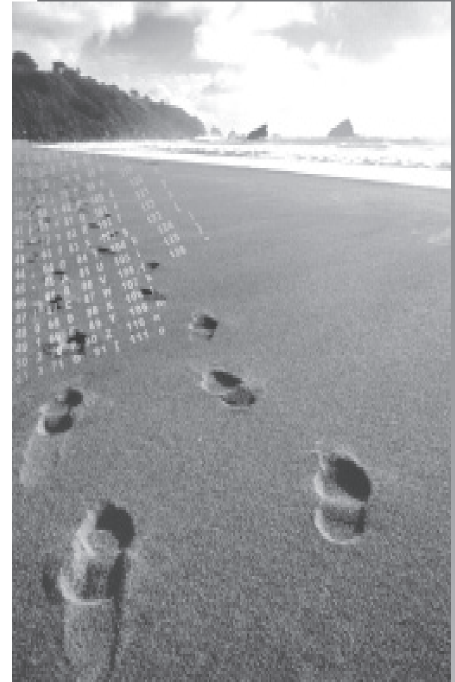
2. ¿Qué diferencia hay entre memoria principal y memoria auxiliar en un computador?

3. ¿Cuáles son los componentes de la CPU de un computador?

4. Describa las funciones de cada uno de los componentes de la CPU de un computador.

Módulo 2

Elaboración de algoritmos y representación de datos



Introducción

En este módulo se presentan los pasos que se siguen en la elaboración de algoritmos, explicando las acciones que deben efectuarse en cada uno de ellos y los elementos necesarios para construir algoritmos. Se presenta además la jerarquía de conceptos en la representación de datos, explicando en detalle cada uno de dichos conceptos con sus correspondientes códigos y usos.

Objetivos del módulo

1. Identificar los pasos que se siguen en la construcción de un algoritmo.
2. Reconocer la jerarquía de representación de datos en un computador.

Preguntas básicas

1. ¿Cuáles son los pasos en la construcción de un algoritmo?
2. ¿Cuáles son los elementos con los que se construye un algoritmo?
3. ¿En qué consiste el análisis de un problema?
4. ¿Qué es una prueba de escritorio?
5. ¿Cuál es la jerarquía de representación de datos en un computador?

Contenidos del módulo

2.1 Pasos en la construcción de un algoritmo
2.2 Elementos para la construcción de un algoritmo
2.3 Representación de datos en un computador

«Reconocer que toda tarea que realizamos en la vida diaria está conformada por una serie de pasos o actividades ejecutadas secuencialmente es el primer paso para comprender cómo se programa un computador. Suministrar los elementos con los cuales se ejecuta una tarea y las acciones que se efectúan con los diferentes elementos es la analogía perfecta para plantear lo que es un algoritmo».

Vea en el botón **Algoritmia básica** del mapa conceptual el video "Módulo 2. Elaboración de algoritmos".

2.1 Pasos en la construcción de un algoritmo

Los pasos que se siguen en la construcción de un algoritmo son:

1. Análisis del problema.
2. Diseño de la solución.
3. Construcción del algoritmo.
4. Prueba de escritorio.

El *análisis* consiste en determinar exactamente cuáles son los datos de entrada que se requieren, cuál es la información que se desea producir y cuál es el proceso que se debe efectuar sobre los datos de entrada para producir la información requerida. Se debe indagar por todas las situaciones especiales que se puedan presentar para tenerlas en cuenta en el diseño.

Con base en el análisis se elabora el *diseño* del algoritmo: se asignan nombres a las variables, se define el tipo de cada una de ellas, se definen las operaciones y/o subprocesos que hay que efectuar y el método para resolver cada uno de ellos.

Los elementos para la *construcción* de un algoritmo son: datos, estructuras e instrucciones. Más adelante hablaremos en detalle de cada uno de ellos.

La *prueba de escritorio* consiste en asumir la posición del computador y ejecutar el algoritmo que se ha elaborado para ver cómo es su funcionamiento. Esta parte es muy importante puesto que permite detectar errores de lógica sin haber usado aún el computador. Aunque no garantiza que el algoritmo está bueno en el 100%, ayuda mucho en la elaboración de algoritmos correctos.

Habiendo superado los pasos anteriores, se elige un lenguaje de programación, se codifica el algoritmo en dicho lenguaje y se pone en ejecución en el computador disponible.

2.2 Elementos para la construcción de un algoritmo

Los elementos con los cuales se construye un algoritmo son las *estructuras lógicas* y *los datos*. Comencemos con los datos.

Para efectos de representación de datos en un computador, éstos se clasifican en *numéricos* y *no numéricos*, y los datos numéricos se clasifican en *enteros* y *reales*.

En términos de computación esta clasificación se denomina **tipo**, y se habla entonces de datos de tipo entero, de tipo real, de tipo no numérico, etc.

Cuando se trabajan datos numéricos en un computador es muy importante considerar si el tipo es entero o real, puesto que, dependiendo de ello, los resultados que se obtienen al efectuar operaciones aritméticas pueden variar sustancialmente. Cuando veamos la evaluación de expresiones haremos notar esta diferencia.

2.3 Representación de datos en un computador

La unidad básica de representación de datos es el **bit**.

Bit es un acrónimo de las palabras en inglés **bi**nary digi**t** (dígito binario). Realmente un bit es un elemento biestable, es decir, un elemento que sólo puede tener dos estados. Por ejemplo, un bombillo sólo puede tener dos estados: encendido o apagado. Por convención se ha establecido que un estado representa el número 1 y el otro estado representa el 0, y con base en esta convención se ha construido la representación de datos en un computador.

La siguiente unidad se denomina ***byte***.

Un ***byte*** es un conjunto de ocho bits.

Si tenemos un conjunto de ocho bits podemos manejar diferentes combinaciones de unos y ceros. Veamos dos ejemplos (figura 2.1):

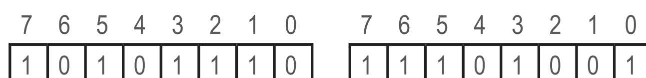


Figura 2.1. Dos combinaciones diferentes de ocho bits.

A cada combinación de unos y ceros que se tenga, se ha establecido que representa un carácter específico. Este sistema de codificación es el que se denomina código **ASCII** (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange).

Los bits se numeran de derecha a izquierda, comenzando con el bit 0 y terminando con el bit 7.

De los ocho bits, sólo siete se utilizan para representar datos; el bit 7 (o sea el octavo) se utiliza para controlar transmisión de datos en la memoria del computador.

Si consideramos cada conjunto de ocho bits como un número binario, éste tendrá su correspondiente valor en el sistema decimal.

A continuación presentamos un subconjunto de códigos ASCII (figura 2.2). El número de la izquierda es el número decimal correspondiente a una combinación de unos y ceros en el *byte*, y el carácter al frente, el carácter que representa según el código ASCII.

Por consiguiente, cada carácter dentro del computador requiere un *byte*.

La siguiente unidad de representación es el **campo**.

Un **campo** es un conjunto de bytes. Si queremos representar el nombre “pedro”, se requieren cinco *bytes*: uno para la “p”, otro para la “e”, otro para la “d”, otro para la “r” y otro para la “o”. Ese conjunto de *bytes* que se requieren para representar algún dato es lo que se denomina campo.

32		52	4	72	H	92	\	112	p
33	!	53	5	73	I	93	]	113	q
34	«	54	6	74	J	94	^	114	r
35	#	55	7	75	K	95	_	115	s
36	$	56	8	76	L	96	‘	116	t
37	%	57	9	77	M	97	a	117	u
38	&	58	:	78	N	98	b	118	v
39	‘	59	;	79	O	99	c	119	w
40	(	60	<	80	P	100	d	120	x
41	)	61	=	81	Q	101	e	121	y
42	*	62	>	82	R	102	f	122	z
43	+	63	?	83	S	103	g	123	{
44	,	64	@	84	T	104	h	124	|
45	-	65	A	85	U	105	i	125	}
46	.	66	B	86	V	106	j	126	~
47	/	67	C	87	W	107	k		
48	0	68	D	88	X	108	l		
49	1	69	E	89	Y	109	m		
50	2	70	F	90	Z	110	n		
51	3	71	G	91	[	111	o		

Figura 2.2. Subconjunto de códigos ASCII.

La siguiente unidad de representación es el **registro**.

Un **registro** es un conjunto de campos. Si queremos describir a una persona, necesitamos varios datos: el nombre, los apellidos, el número de la cédula, la fecha de nacimiento, etc. Para representar cada uno de estos datos se requiere un campo. Hablaríamos del campo de nombre, del campo de apellido, del campo de cédula, del campo de fecha de nacimiento, etc. Ese conjunto de campos con los cuales se representa algún ente se denomina registro.

La siguiente unidad de representación es el **archivo**.

Un **archivo** es un conjunto de registros. Si queremos almacenar la información correspondiente a todos los estudiantes que están viendo el curso de algoritmos, necesitamos un registro para el estudiante Pedro, otro registro para el estudiante Juan, otro registro para el estudiante Pablo, y así sucesivamente. Ese conjunto de registros es lo que se conoce como archivo.

La siguiente unidad de representación es la **base de datos**.

Una **base de datos** es un conjunto de archivos relacionados entre sí. Para poder efectuar, digamos, un proceso de matrícula, se requiere un archivo con los estudiantes, otro archivo con el pénsum de la carrera, otro archivo con los horarios de los cursos, otro archivo con los profesores. Este conjunto de archivos (que deben estar relacionados) para efectuar un proceso de matrícula es lo que se conoce como base de datos.

Resumen

En este módulo hemos presentado los pasos que se siguen en la elaboración de algoritmos y la jerarquía de representación de datos en un computador.

Ejercicios propuestos

1. ¿Cuántos bits se requieren para representar la palabra "paquidermo"?
2. Describa los conceptos de bit, *byte*, campo, registro, archivo y base de datos.
3. Explique la diferencia entre cada uno de los conceptos de la pregunta anterior.

Capítulo 2 Estructuras básicas en el desarrollo de algoritmos

Contenido breve

Módulo 3
Estructuras para la construcción de algoritmos e instrucciones de lectura y escritura
Módulo 4
Instrucción de asignación y expresiones aritméticas
Módulo 5
Manipulación de expresiones algebraicas
Módulo 6
Expresiones relacionales y lógicas
Módulo 7
Aplicaciones del capítulo

En el capítulo anterior se presentó la herramienta con la cual se va a trabajar en el curso y su ubicación en el desarrollo de soluciones. En este capítulo se introduce el concepto de programación, el cual se manifiesta esencialmente en los algoritmos.

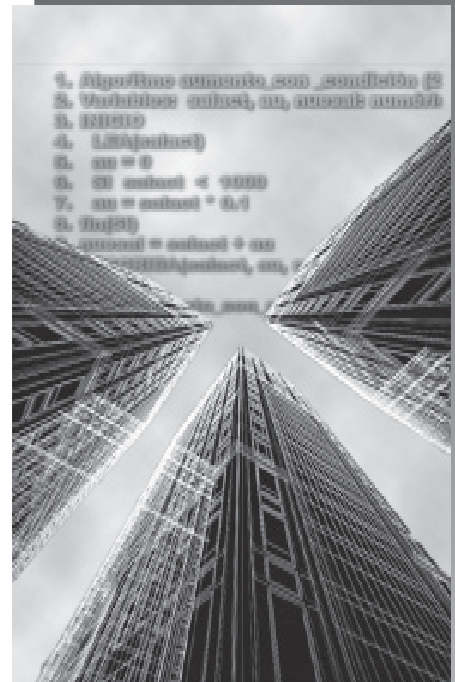
Ejecutar nuestras tareas una por una hasta terminarlas es una buena estrategia para no tener labores pendientes o empezadas.

El concepto de algoritmo, como un conjunto de instrucciones con sus datos, constituye la parte fundamental del capítulo.

En este capítulo se estudian los importantes conceptos de datos, constantes, variables, expresiones, instrucciones de asignación, instrucciones de entrada e instrucciones de salida.

Módulo 3

Estructuras para la contrucción de algoritmos e instrucciones de lectura y escritura



«La construcción de un edificio requiere un conjunto de estructuras y materiales (fundaciones, columnas, vigas, losas, puertas, ventanas, etc.), los cuales, al ensamblarse, generan un producto final: una edificación. La construcción de un algoritmo requiere datos y estructuras lógicas».

Introducción

En este módulo se presentan las estructuras básicas para la construcción de algoritmos (cada una de ellas con sus respectivas instrucciones), la forma general en que se escribirán los algoritmos que se elaboren en este texto, la instrucción de lectura y la instrucción de escritura.

Objetivos del módulo

1. Identificar las estructuras lógicas, con sus correspondientes instrucciones, para la construcción de algoritmos.
2. Reconocer la forma general de un algoritmo.
3. Aplicar la instrucción de lectura.
4. Aplicar la instrucción de escritura.

Preguntas básicas

1. ¿Cuáles son las estructuras lógicas necesarias para construir un algoritmo?
2. ¿En qué consiste la estructura secuencia?
3. ¿Cuál es la forma general de un algoritmo?
4. ¿En qué consiste la instrucción de lectura?
5. ¿En qué consiste la instrucción de escritura?

Contenidos del módulo

3.1 Estructuras lógicas para la construcción de algoritmos
3.2 Forma general de un algoritmo
3.3 Instrucción de lectura
3.4 Instrucción de escritura

Vea en los botones **Lectura** y **Escritura** del mapa conceptual el video "Módulo 3. Estructuras".

3.1 Estructuras lógicas para la construcción de algoritmos

Las estructuras lógicas para la construcción de algoritmos son:

1. Estructura secuencia.
2. Estructura decisión.
3. Estructura ciclo.

La estructura básica en la construcción de un algoritmo es la *estructura secuencia*. Esta estructura consiste en que las instrucciones se ejecutan exactamente en el orden en que han sido escritas: primero se ejecuta la primera instrucción, luego la segunda instrucción, luego la tercera instrucción y por último la última instrucción.

> *El orden en el cual se escriben las instrucciones es fundamental para el correcto funcionamiento de un algoritmo.*

Cada estructura consta de un conjunto de instrucciones.

Las instrucciones correspondientes a la estructura secuencia son:

1. Instrucciones de lectura.
2. Instrucciones de escritura.
3. Instrucciones de asignación.
4. Las instrucciones correspondientes a la estructura decisión.
5. Las instrucciones correspondientes a la estructura ciclo.

Las instrucciones correspondientes a la *estructura decisión* son:

1. La instrucción SI y su componente opcional DE_LO_CONTRARIO.
2. La instrucción CASOS.

Las instrucciones correspondientes a la estructura ciclo son:

1. La instrucción MIENTRAS.
2. La instrucción PARA.
3. La instrucción HAGA.

3.2 Forma general de un algoritmo

La forma general de nuestros algoritmos será:

```
Algoritmo nombreDelAlgoritmo
  Definición de variables
  INICIO
       Instrucciones del programa
  FIN
Fin(nombreDelAlgoritmo)
```

3.3 Instrucción de lectura

Para que el computador pueda procesar datos, éstos deben estar en la memoria principal (RAM). La instrucción de lectura consiste en llevar los datos con los cuales se desea trabajar, desde un medio externo hacia la memoria principal.

Los medios externos en los cuales pueden residir los datos son: disco duro, disco removible, disquete, cinta, etc.; los datos también pueden entrarse directamente a través del teclado.

La forma general de la instrucción de lectura es:

LEA(lista de variables, separadas por comas)

En nuestro texto, en los primeros algoritmos consideraremos que los datos entran directamente desde el teclado.

3.4 Instrucción de escritura

La instrucción de escritura consiste en llevar datos desde la memoria hacia un medio externo, el cual puede ser un disco duro, una cinta, una impresora, etc.

La forma general de la instrucción de escritura es:

ESCRIBA(lista de variables y/o mensajes, separados por comas)

Los mensajes son para instruir al usuario acerca de los datos que se le están presentando. Si el dato que se imprime es la estatura de una persona, es conveniente que esté precedido por un mensaje que diga: estatura. Si el dato que se está presentando es una edad, es conveniente que esté precedido por un mensaje que diga: edad. Y así sucesivamente.

Cuando vayamos a escribir un mensaje en una instrucción de lectura, dicho mensaje lo escribiremos encerrado entre comillas.

Vamos a hacer nuestro primer algoritmo utilizando sólo las instrucciones de lectura y de escritura.

```
1.   Algoritmo uno
2.       Variables: nom: alfanumérica
3.                  tel: numericaEntera
4.       INICIO
5.           ESCRIBA(“teclee nombre y teléfono”)
6.           LEA(nom, tel)
7.           ESCRIBA(“nombre: “, nom, “ teléfono: “, tel)
8.       FIN
9.   Fin(uno)
```

En la instrucción 1 estamos definiendo el nombre del algoritmo: lo estamos llamando *uno*.

En la instrucción 2 estamos definiendo las variables que vamos a utilizar en nuestro programa. Ponemos el título *variables* y definimos la variable **nom**, que podrá almacenar datos alfanuméricos, y en la instrucción 3 definimos la variable **tel**, que podrá almacenar datos numéricos enteros.

En la instrucción 4 ponemos nuestra palabra clave INICIO, la cual indica que a partir de ahí están las instrucciones de nuestro algoritmo.

En la instrucción 5 ponemos nuestra instrucción de escritura, la cual escribe el mensaje «teclee nombre y teléfono» instruyendo al usuario acerca de los datos que debe teclear.

En la instrucción 6 ponemos nuestra instrucción de lectura, en la cual los datos tecleados por el usuario se almacenarán en las posiciones de memoria que el computador identificará con los nombres **nom** y **tel**.

En la instrucción 7 ponemos la instrucción de escritura con la cual se escriben los datos tecleados por el usuario, cada uno con su respectivo título.

La instrucción 8 es nuestra palabra clave FIN, que cierra el inicio de las instrucciones del algoritmo, y en la instrucción 9 ponemos el fin del algoritmo.

Resumen

En este módulo se han definido las estructuras lógicas para la construcción de algoritmos y las instrucciones de entrada y salida de datos en un computador.

Ejercicios propuestos

1. ¿Cómo se diferencia la escritura de mensajes de la escritura de datos en una instrucción de escritura?

2. Detecte y describa los errores que hay en el siguiente algoritmo. Proponga soluciones.

```
Algoritmo jabon
        Variables: a, b: numericaEntera.
        INICIO
                ESCRIBA(“teclee los datos para a y b: “)
                ESCRIBA(“dato a: “, a, “ dato b: “, b)
                LEA(a, b)
        FIN
Fin(jabon)
```

3. ¿Cuáles son las instrucciones correspondientes a la estructura ciclo?

4. ¿Cuáles son las instrucciones correspondientes a la estructura decisión?

5. ¿Cuáles son las instrucciones correspondientes a la estructura secuencia?

Módulo 4

Instrucción de asignación y expresiones aritméticas

Introducción

En este capítulo se tratará lo correspondiente a la instrucción de asignación, la cual consiste en llevar datos hacia alguna posición de la memoria principal del computador, y todo lo correspondiente a las expresiones aritméticas: construcción, operadores, tipos y evaluación.

Objetivos del módulo

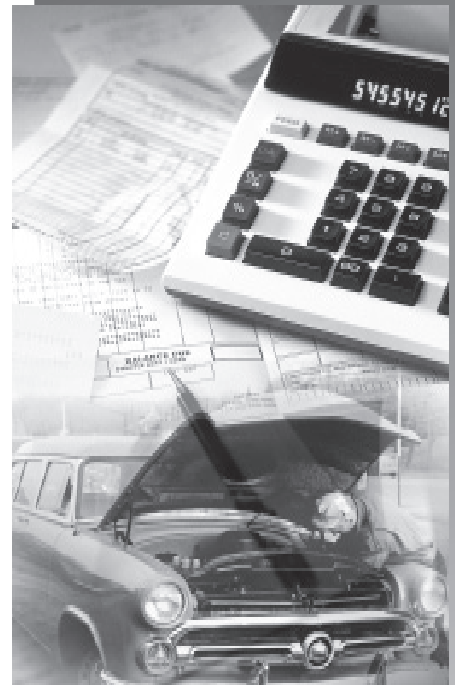
1. Emplear la instrucción de asignación.
2. Construir y manipular expresiones aritméticas.

Preguntas básicas

1. ¿En qué consiste una instrucción de asignación?
2. ¿Cómo se evalúa una expresión aritmética?
3. ¿Cuál es el orden de evaluación de los operadores en una expresión aritmética?
4. ¿En qué consiste la asociatividad de un operador?
5. ¿Cómo afecta el tipo de operando el resultado de evaluar una expresión?

Contenidos del módulo

4.1 Instrucción de asignación
4.2 Expresiones aritméticas



«Asignar consiste en suministrar recursos a algo o a alguien. Cuando en una empresa se contrata a un empleado, a éste se le asigna un salario. Con base en este salario se determina el pago del empleado. En algunas situaciones el cálculo del pago implica efectuar operaciones aritméticas: el pago es el valor de la hora multiplicado por las horas trabajadas, más el número de horas extras por el valor de la hora, más las horas nocturnas por un recargo, etc».

Vea en los botones **Asignación** y **Aritméticas** del mapa conceptual el video "Módulo 4. Asignación".

4.1 Instrucción de asignación

La instrucción de asignación consiste en llevar algún dato a una posición de memoria, la cual está identificada con el nombre de una variable.

La forma general de una instrucción de asignación es:

Variable = {
- constante numérica entera
- constante numérica real
- variable
- mensaje
- expresión

Ejemplos:

1. a = 316
2. b = 3.14
3. c = «hola mundo»
4. d = a
5. e = a + b * a

En los ejemplos 1 y 2, a las variables **a** y **b** les estamos asignando una constante numérica: entera en el primer ejemplo, real en el segundo.

En el ejemplo 3, a la variable **c** le estamos asignando un mensaje.

En el ejemplo 4, a la variable **d** le estamos asignando el contenido de otra variable.

En el ejemplo 5, a la variable **e** le estamos asignando el resultado de evaluar una expresión.

Vamos a ocuparnos de lo que son expresiones.

4.2 Expresiones aritméticas

En general, una expresión es una sucesión de operandos y operadores, la cual puede ser de tres clases:

1. Expresión aritmética.
2. Expresión relacional.
3. Expresión lógica.

Comencemos tratando lo que son las expresiones aritméticas:

Una *expresión aritmética* es una sucesión de operandos y operadores aritméticos, en la cual los operadores actúan sobre los operandos.

Los operadores aritméticos son (tabla 4.1):

Tabla 4.1. Operadores aritméticos y su significado.

Operador	Operación
^	Potenciación
*	Multiplicación
/	División
%	Módulo
+	Suma
–	Resta

De los operadores mostrados vale la pena indicar en qué consiste el operador módulo (%).

El operador módulo calcula el residuo de una división entera.

Si tenemos la instrucción

a = 12 % 7

en la posición de memoria identificada con la variable **a** quedará almacenado el valor de 5.

Cuando se escribe una expresión aritmética es necesario tener en cuenta las características de evaluación de dicha expresión. Dichas características incluyen lo que es la prioridad de ejecución de los operadores y la asociatividad de cada uno de ellos. La *prioridad* se refiere a cuáles operaciones se ejecutan primero y cuáles de últimas, y la *asociatividad* se refiere al orden en el cual se ejecutan operaciones consecutivas con la misma prioridad.

En expresiones aritméticas de computador la prioridad de ejecución de las operaciones es la siguiente:

1. Primero se ejecutan las operaciones de potenciación.
2. En segundo lugar se ejecutan las operaciones de multiplicación, división y módulo.
3. En tercer lugar se ejecutan las operaciones de sumas y restas.

La asociatividad se refiere a:

1. Cuando se hallan operaciones consecutivas de potenciación, éstas se ejecutan de derecha a izquierda, es decir, la operación de potenciación es asociativa por la derecha.
2. Las demás operaciones son asociativas por la izquierda, es decir, se ejecutan en el orden que aparezcan, de izquierda a derecha. En otras palabras, las operaciones multiplicación, división, módulo, suma y resta son asociativas por la izquierda.

Considerando estas dos características vamos a presentar algunos ejemplos que las ilustren.

Consideremos que en las variables **a**, **b**, **c**, **d**, **e** y **f** están almacenados los siguientes datos (tabla 4.2):

Tabla 4.2. Variables con sus valores.

Variable	Valor
a	4
b	5
c	6
d	3
e	2
f	4

y consideremos la expresión

a + b * c

Al evaluar la expresión, primero ejecuta el producto b * c, y se obtiene como resultado 30.

La expresión queda:

a + 30

Luego ejecuta la suma y el resultado es 34.

Consideremos algunos ejemplos:

1. Si tenemos la expresión

   a + b * c / d – e ^ f

   el resultado de evaluar dicha expresión, considerando los valores mostrados en la tabla 4.2, es:

   primero ejecuta **e ^ f**, y la expresión queda:

   a + b * c / d – 16

   luego ejecuta **b * c**, y la expresión queda:

   a + 30 / d – 16

   luego ejecuta **30 / d**, y la expresión queda:

   a + 10 – 16

   luego ejecuta **a + 10**, y la expresión queda:

   14 – 16

   por último ejecuta **14 – 16**, y se obtiene un resultado de **–2.**

2. Consideremos ahora una expresión con operaciones de multiplicación consecutivas:

    a * b * c

    primero ejecuta **a * b**, y la expresión queda:

    20 * c

    luego ejecuta **20 * c**, y se obtiene como resultado **120.**

3. Consideremos que tenemos esta otra expresión con operaciones de potenciación consecutivas:

    a ^ d ^ e

    primero ejecuta **d ^ e**, y la expresión queda:

    a ^ 9

    luego ejecuta **a ^ 9** y obtiene **262144.**

El orden de ejecución de las operaciones, con base en la prioridad y la asociatividad, se puede alterar mediante el uso de paréntesis.

Consideremos que la expresión del ejemplo 3 la escribimos así:

(a ^ d) ^ e

El resultado de evaluarla sería:

primero ejecuta la operación entre paréntesis, y la expresión queda

(64) ^ e

la cual al resolverse da un resultado de **4096**.

En general, cuando la expresión se escribe utilizando paréntesis, primero se resuelven todas las operaciones que están dentro del paréntesis, desde adentro hacia fuera, y luego se aplica el orden de ejecución de acuerdo a la prioridad y la asociatividad de los operadores.

Es supremamente importante tener en cuenta estas características de prioridad y de asociatividad a la hora de escribir expresiones, puesto que afectará notablemente el resultado de la evaluación.

Consideremos ahora cómo afecta el tipo de los operandos el resultado de evaluar una expresión. Cuando se efectúa alguna operación entre operandos de tipo entero el resultado que se obtiene es también de tipo entero. Dada esta característica, si efectuamos la instrucción

a = 7 / 5

el resultado obtenido es **1.**

Si quisiéramos que el resultado tuviera decimales deberíamos escribir

a = 7. / 5 o
a = 7 / 5. o
a = 7. / 5.

En cualquiera de estas tres expresiones, el resultado que se asigna a **a** es **1.4.**

Resumen

Hemos visto en qué consiste la instrucción de asignación, el manejo de expresiones y la forma de evaluarlas, y cómo la evaluación se afecta de acuerdo con la prioridad de ejecución de los operadores, la asociatividad y el tipo de los operandos.

Ejercicios propuestos

1. Dada la siguiente definición de variables con su respectivo tipo y contenido:

Numéricas enteras	Numéricas reales
a = 3	x = 2.
b = 5	y = 3.5
c = 2	z = 5.
d = 4	w = 1.2

determine el resultado de evaluar cada una de las siguientes expresiones:

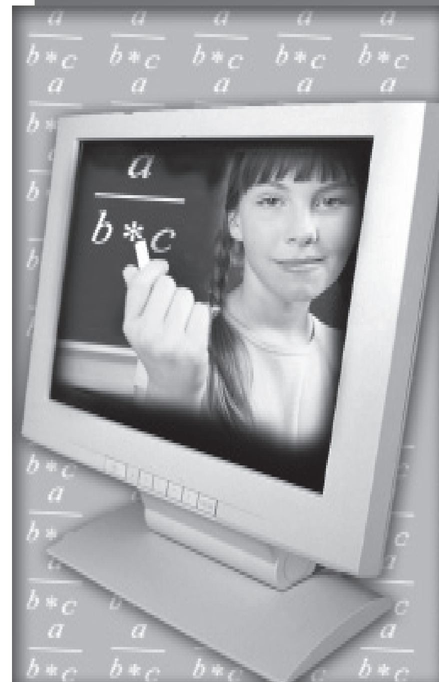
1. a * b / 2 + 1
2. c / x ^ a
3. a / (b + c / (d + 1) * (a + b) – a) ^ b ^ a + c
4. x / (b + c / (y + 1) * (a + b) – a) ^ b ^ a + w
5. a + (b – c * (d – (a + b) * (c – c / d * a) / a + b * c) – (b ^ c)) ^ b
6. z / x + b * w * (c – b) / a

Módulo 5

Manipulación de expresiones algebraicas

Introducción

En el módulo anterior se trataron las expresiones aritméticas como parte de la instrucción de asignación y se estudiaron las características que tienen dichas expresiones. En este módulo se verá la relación que existe entre lo que son las expresiones aritméticas de computador frente a las expresiones algebraicas, cómo se establece la correspondencia entre ellas y cómo convertir una expresión de computador en expresión algebraica y viceversa.



«En el campo de las matemáticas se tiene establecido todo un formalismo para escribir y evaluar expresiones aritméticas y expresiones algebraicas. Cuando se trata de instruir a un computador para que evalúe alguna de estas expresiones hay que establecer una correspondencia entre una expresión algebraica y una expresión de computador. El éxito de un programa de computador depende de que dicha correspondencia se efectúe bajo ciertas condiciones que están previamente establecidas».

Objetivos del módulo

1. Construir una expresión lineal de computador con base en una expresión algebraica.
2. Construir una expresión algebraica con base en una expresión lineal de computador.

Preguntas básicas

1. ¿Para qué se utilizan paréntesis en expresiones de computador?
2. ¿Cómo se construye una expresión de computador para operaciones de potenciación?

Contenidos del módulo

5.1 Conversión de una expresión algebraica a expresión lineal de computador
5.2 Conversión de una expresión lineal de computador a expresión algebraica

Vea en el botón **Aritméticas** del mapa conceptual el video "Módulo 5. Conversiones".

5.1 Conversión de una expresión algebraica a expresión lineal de computador

Cuando se elabora un algoritmo es muy común tener que escribir expresiones, sobre todo si se trata de algoritmos de carácter científico o matemático.

Para escribir expresiones de computador, según el módulo anterior, es necesario tener en cuenta la forma como el computador evalúa dichas expresiones.

Vamos a ver cómo convertir una expresión algebraica en expresión de computador. Comencemos considerando un primer ejemplo. Sea la expresión algebraica

$$\frac{a}{b * c}.$$

Si queremos escribir esta expresión algebraica como expresión de computador, tenemos varias formas de hacerlo:

1. a / b * c
2. a / b / c
3. a / (b * c)

La primera forma es *incorrecta,* porque de acuerdo a lo visto en el módulo anterior, primero ejecuta la división del valor de **a** entre el valor de **b**, y el resultado lo multiplica por el valor de **c**. O sea que si **a** vale 36, **b** vale 6 y **c** vale 2 el resultado de evaluar dicha expresión es 12, lo cual es erróneo.

La segunda y tercera formas son *correctas*: en la segunda forma, primero ejecuta la división del valor de **a** por el valor de **b** y el resultado lo divide por el valor de **c**, obteniendo como resultado 3. En la tercera forma primero multiplica el valor de **b** por el valor de **c** y el resultado divide al valor de **a**, obteniendo como resultado también 3.

Es supremamente importante entender este primer ejemplo. En la tercera forma hemos utilizado paréntesis para alterar el orden de ejecución de las operaciones; sin embargo, en la segunda forma no los hemos utilizado y el resultado también es correcto.

Pasemos a considerar ejemplos más complejos, algunos de los cuales exigen el uso de paréntesis.

1. Veamos la siguiente expresión algebraica:

$$\frac{a + \frac{d}{e} + c}{b * c}.$$

   La expresión de computador correcta puede ser:

   1. (a + d / e + c) / b / c   o
   2. (a + d / e + c) / (b * c)

2. Veamos esta otra expresión algebraica

$$a^{b^{c+1}}$$

La forma correcta de escribir esta expresión algebraica como expresión de computador es:

a ^ b ^ (c + 1)

5.2 Conversión de una expresión lineal de computador a expresión algebraica

Pasemos ahora a considerar el caso contrario: dada una expresión de computador, escribir la expresión algebraica correspondiente.

Veamos los siguientes ejemplos:

1. a + b * c / d – e ^ f

   La expresión algebraica es:

   $$a + \frac{b * c}{d} - e^f.$$

2. (a + b) * c / (d – e) ^ f

   La expresión algebraica es:

   $$\frac{(a + b) * c}{(d - e)^f}.$$

3. a – b / c + (b – c / d) / e

   La expresión algebraica es:

   $$a - \frac{b}{c} + \frac{-\frac{c}{d}}{e}.$$

Resumen

En este módulo se ha tratado lo referente a la conversión de expresiones algebraicas a expresiones lineales de computador y viceversa.

Ejercicios propuestos

1. Convierta las siguientes expresiones algebraicas en expresiones de computador:

   a. $\dfrac{a - \dfrac{b+c}{d-e} + e - f^{2.3}}{4.5 + d}$

   b. $(3.14)^{(a+b)} + \dfrac{b^{*} c^{a+b}}{a+b} - a * b$

   c. $\dfrac{a + \dfrac{b-c}{d-a} - \dfrac{c}{d}}{a+b} - \dfrac{\dfrac{c}{d+a} + b^{c^{a}}}{\dfrac{d+1}{3+a}}$

2. Convierta las siguientes expresiones de computador en expresiones algebraicas:

   a. a / (b + c / (d + 1) * (a + b) – a) ^ b ^ a + c
   b. a / b / c / d ^ a * b – d / a * c
   c. a + (b – c * (d – (a + b) * (c – e / d * a) / a + b * c) – (e ^ f)) ^ b

Módulo 6

Expresiones relacionales y lógicas

Introducción

En este módulo se tratará lo correspondiente a expresiones relacionales y expresiones lógicas, las cuales, al ser evaluadas, arrojan un resultado lógico, es decir, verdadero o falso. Con base en estas expresiones se podrán detectar diferentes situaciones que se presentan cuando se construyen algoritmos y se podrá instruir a la máquina para que ejecute las operaciones apropiadas según el resultado de una condición.

Objetivos del módulo

1. Construir y manipular expresiones relacionales.
2. Construir y manipular expresiones lógicas.

Preguntas básicas

1. ¿Qué es una expresión relacional?
2. ¿Qué resultado se obtiene al evaluar una expresión relacional?
3. ¿Cuáles son los operadores relacionales?
4. ¿Qué es una expresión lógica?
5. ¿Cuáles son los operadores lógicos?

Contenidos del módulo

6.1 Expresiones relacionales
6.2 Expresiones lógicas

«La toma de decisiones es un asunto importante en el desarrollo de todo proyecto. Tomar una decisión se hace con base en situaciones lógicas o que implican relación. Si comparo el costo de dos materiales para tomar una decisión, estoy estableciendo una relación, pero si lo que evalúo es una situación como saber si está de día o de noche, estoy tomando una decisión con base en una situación lógica: está de día si hay luz solar, de lo contrario está de noche».

Vea en los botones **Relacionales** y **Lógicas** del mapa conceptual el video "Módulo 6. Expresiones relacionales".

6.1 Expresiones relacionales

Las expresiones relacionales producen un resultado lógico: verdadero o falso. Se construyen con base en operandos y operadores relacionales.

Operandos:
- Variables
- Constantes
- Expresiones aritméticas

Operadores relacionales:

>	Mayor que
>=	Mayor o igual que
<	Menor que
<=	Menor o igual que
==	Igual
≠	Diferente

Cuando se trata de evaluar una expresión relacional primero se evalúan las expresiones aritméticas y luego se evalúan los operadores relacionales.

Ejemplos de expresiones relacionales:

1. a > b
2. 3.14 <= b + c
3. (b – c) / 2.54 == a – b * c / d

En el ejemplo 1 simplemente compara el valor almacenado en **a** con el almacenado en **b**. Si el valor almacenado en **a** es mayor que el almacenado en **b,** el resultado de evaluar la expresión es verdadero; de lo contrario es falso.

En el ejemplo 2 primero ejecuta la suma de **b** con **c**, y el resultado lo compara con 3.14. Si 3.14 es menor o igual que la suma de **b** con **c**, el resultado de evaluar la expresión es verdadero; de lo contrario es falso.

En el ejemplo 3 primero evalúa la expresión aritmética del lado izquierdo del operador == de acuerdo al orden de evaluación de expresiones aritméticas, luego la expresión aritmética del lado derecho del operador ==, y por último compara los resultados obtenidos. Si dichos resultados son iguales, el resultado de evaluar la expresión será verdadero; de lo contrario, falso.

6.2 Expresiones lógicas

Son expresiones que también producen un resultado lógico. Están construidas con operandos y operadores lógicos.

Operandos:
- Expresiones relacionales
- Variables lógicas

Operadores lógicos:

!	Negación
&&	Conjunción
||	Disyunción

Variables lógicas son variables que sólo pueden tomar dos valores: verdadero o falso.

En general, una variable lógica, en el ambiente de computadores, es una variable de un solo bit, el cual puede ser 0 o 1. Por convención se ha adoptado que el 0 representa falso y el 1 verdadero.

Cuando se trata de evaluar expresiones lógicas primero se evalúan las expresiones aritméticas, luego las expresiones relacionales y por último los operadores lógicos, los cuales también tienen cierta prioridad en el momento de efectuar la evaluación.

Prioridad de los operadores lógicos:

1. Negación.
2. Conjunción.
3. Disyunción.

Ejemplos de expresiones lógicas:

1. a && b
2. a > b || c < d
3. 3.14 * radio >= a ^ 2 || (b – c) == (3.1 + c) && (c + d) * 2 <= 1

En el ejemplo 1, si **a** y **b** tienen estado de verdad, el resultado de evaluar la expresión es verdadero; de lo contrario es falso.

En el ejemplo 2, si **a** es mayor que **b**, o **c** es menor que **d**, el resultado de evaluar la expresión es verdadero; de lo contrario es falso.

En el ejemplo 3, primero se evalúan las expresiones aritméticas:

Llamemos **r1** el resultado de multiplicar 3.14 por el valor almacenado en la variable **radio**.
Llamemos **r2** el resultado de elevar el contenido de la variable **a** al cuadrado.
Llamemos **r3** el resultado de restarle a **b** lo que hay almacenado en **c**.
Llamemos **r4** el resultado de sumar 3.1 con el contenido de **c**.
Llamemos **r5** el resultado de multiplicar por 2 la suma de **c** con **d**.

Nuestra expresión quedará:

r1 >= r2 || r3 == r4 && r5 <= 1

Llamemos **r6** el resultado lógico obtenido de comparar **r1** con **r2**.
Llamemos **r7** el resultado de comparar **r3** con **r4**.
Llamemos **r8** el resultado de comparar **r5** con 1.

Nuestra expresión queda:

r6 || r7 && r8

Luego evalúa **r7** && **r8**

Llamemos **r9** este resultado: si **r7** y **r8** son verdaderos, entonces **r9** será verdadero; de lo contrario **r9** será falso.

Nuestra expresión queda:

r6 || r9

en la cual, con uno de los dos operandos que sea verdadero, el resultado de evaluar la expresión será verdad.

Resumen

En este módulo hemos tratado el manejo y la evaluación de expresiones relacionales y lógicas.

Ejercicios propuestos

1. Dada la siguiente definición de variables con su respectivo tipo y contenido:

Numéricas enteras	Numéricas reales	Lógicas
a = 3	x = 2.	m = T
b = 5	y = 3.5	n = F
c = 2	z = 5.	o = F
d = 4	w = 1.2	p = T

donde T es verdad y F es falso, determine el resultado de evaluar cada una de las siguientes expresiones:

1. a + b ≠ 8 ^ d * a / c + 2
2. a + b ≠ 8 ^ d * (a / c) + 2
3. m && n || m && p
4. z / 2 < 2.5
5. m && (n || m) && p
6. c * d – b ^ c >= y * x / b || m && p && a + b > x + y

Módulo 7

Aplicaciones del capítulo

Introducción

En los módulos anteriores se han presentado los conceptos básicos elementales para elaborar algoritmos. En este módulo se presentan dos ejemplos en los cuales se aplican conjuntamente los conceptos tratados.

Objetivos del módulo

1. Aplicar la estructura secuencia con sus instrucciones básicas: lectura, escritura y asignación.

Preguntas básicas

1. ¿Cómo se aplica la estructura secuencia?
2. ¿Para qué son los mensajes de la instrucción ESCRIBA en un algoritmo?
3. Cuál instrucción se ejecuta primero: ¿LEA o ESCRIBA?

Contenidos del módulo

7.1 Algoritmo para calcular el área de un cuadrado
7.2 Algoritmo para determinar el nuevo salario de dos empleados

«La mejor manera de apropiarse de un conjunto de conceptos es viendo la aplicación de ellos. La elaboración de un algoritmo comprende los pasos de análisis, definición de variables (cuáles son de entrada, cuáles son de salida) y construcción de las instrucciones para llevar a cabo la tarea propuesta».

Vea en los botones **Secuencia** y **Datos** del mapa conceptual el video “Módulo 7. Elaboración de algoritmos”.

7.1 Algoritmo para calcular el área de un cuadrado

Elaborar un algoritmo que lea un dato correspondiente al lado de un cuadrado y que calcule e imprima el área de dicho cuadrado.

Análisis:

1. Datos de entrada:
    - Lado del cuadrado: lado

2. Datos de salida:
    - Área del cuadrado: area

3. Cálculos:
    - Determinar el área del cuadrado aplicando la fórmula lado al cuadrado.

Nuestro algoritmo será:

```
1.   Algoritmo areaDelCuadrado
2.        Variables: lado, area: numericaEntera
3.        INICIO
4.             LEA(lado)
5.             area = lado ^ 2
6.             ESCRIBA(“lado: “, lado, “ área: “, area)
7.        FIN
8.   Fin(areaDelCuadrado)
```

En la instrucción 1 definimos el nombre del algoritmo.

En la instrucción 2 definimos las variables que vamos a utilizar, con su correspondiente tipo.

En la instrucción 3 ponemos nuestra palabra clave de inicio de las instrucciones ejecutables.

En la instrucción 4 leemos el valor del lado.

En la instrucción 5 determinamos el área del cuadrado y la guardamos en la variable llamada **area**.

En la instrucción 6 escribimos el lado leído y el área calculada, con sus respectivos mensajes.

En la instrucción 7 escribimos nuestra palabra clave de fin de instrucciones ejecutables.

En la instrucción 8 indicamos el fin del algoritmo.

Si el lado leído en la instrucción 4 tiene un valor de 10, al ejecutar la instrucción 5 se almacena en la posición de memoria identificada con el nombre **area** el valor de 100.

Luego, al ejecutar la instrucción 6, imprimirá:

```
lado: 10    área:  100
```

7.2 Algoritmo para determinar el nuevo salario de dos empleados

Elaborar un algoritmo que lea el salario actual de dos empleados y el porcentaje de aumento de cada uno de ellos y que calcule e imprima el salario actual, el aumento y el nuevo salario de cada uno de ellos.

Análisis:

1. Datos de entrada:
   - Salario actual empleado uno: salemp1
   - Salario actual empleado dos: salemp2
   - Porcentaje aumento empleado uno: poremp1
   - Porcentaje aumento empleado dos: poremp2

2. Datos de salida:
   - Aumento empleado uno: auemp1
   - Aumento empleado dos: auemp2
   - Nuevo salario empleado uno: nuesalemp1
   - Nuevo salario empleado dos: nuesalemp2

3. Cálculos:
   - El aumento de cada empleado se obtiene multiplicando el salario actual por el porcentaje de aumento leído, y el nuevo salario se obtiene sumando el aumento al salario actual.

Considerando lo anterior procedemos a escribir el algoritmo que efectúe dicha tarea:

```
1.  Algoritmo aumentos
2.         Variables: salemp1, salemp2, poremp1, poremp2, auemp1, auemp2,
           nuesalemp1, nuesalemp2:  numericaReal.
3.         INICIO
4.              LEA(salemp1, poremp1, salemp2, poremp2)
5.              auemp1 = salemp1 * poremp1
6.              auemp2 = salemp2 * poremp2
7.              nuesalemp1 = salemp1 + auemp1
8.              nuesalemp2 = salemp2 + auemp2
9.              ESCRIBA("empleado uno: ")
10.             ESCRIBA("salario actual: ", salemp1, " aumento: ", auemp1, " nuevo
                         salario: ",  nuesalemp1)
11.             ESCRIBA("empleado dos: ")
12.             ESCRIBA("salario actual: ", salemp2, " aumento: ", auemp2, " nuevo
                         salario: ", nuesalemp2)
13.        FIN
14.   Fin(aumentos)
```

En la instrucción 2 estamos definiendo las variables que vamos a utilizar en nuestro algoritmo, con su respectivo tipo.

En la instrucción 4 leemos los datos correspondientes al salario de cada empleado con su respectivo porcentaje de aumento.

En las instrucciones 5 y 6 calculamos el aumento de cada uno de los empleados: el aumento se obtiene multiplicando el salario actual por el porcentaje leído. Dichos

aumentos los almacenamos en las variables que hemos llamado **auemp1** y **auemp2,** respectivamente.

En las instrucciones 7 y 8 calculamos el nuevo salario de cada empleado: el nuevo salario se obtiene sumándole al salario actual el aumento que se obtuvo en las instrucciones 5 y 6.

En la instrucción 9 escribimos el título para los datos del empleado uno.

En la instrucción 10 escribimos los datos correspondientes al empleado uno: salario actual, aumento y nuevo salario.

En la instrucción 11 escribimos el título para los datos del empleado dos.

En la instrucción 12 escribimos los datos correspondientes al empleado dos: salario actual, aumento y nuevo salario.

Veamos ahora una pequeña prueba de escritorio:

Si los datos entrados son **1000.**, **0.1**, **2000.** y **0.05**, al ejecutar la instrucción 4 los contenidos de las variables leídas son:

salemp1 = 1000.
poremp1 = 0.1
salemp2 = 2000.
poremp2 = 0.05

La ejecución de la instrucción 5

auemp1 = salemp1 * poremp1

hará que en la posición de memoria llamada **auemp1** quede almacenado un valor de 100.

La ejecución de la instrucción 6

auemp2 = salemp2 * poremp2

hará que en la posición de memoria llamada **auemp2** quede almacenado un valor de 100.

Al ejecutar las instrucciones 7 y 8 el contenido de las variables **nuesalemp1** y **nuesalemp2** será 1100 y 2100, respectivamente.

El resultado de ejecutar las instrucciones 9 y 10 producirá el siguiente resultado:

Empleado uno:

Salario actual: 1000   Aumento: 100   Nuevo salario: 1100

Y al ejecutar las instrucciones 11 y 12 el resultado será:

Empleado dos:

Salario actual: 2000 Aumento: 100 Nuevo salario: 2100

Resumen

Hemos presentado un primer gran ejemplo de un algoritmo en el cual se utilizan las herramientas definidas hasta el momento: estructura secuencia, instrucciones de entrada y salida de datos y evaluación de expresiones.

Ejercicios propuestos

1. Elabore un algoritmo que lea el nombre de una persona y que imprima el mensaje 'Hola' seguido del nombre de la persona leída.

2. Elabore un algoritmo que lea el nombre de una persona, la estatura y el peso. La estatura está en centímetros y el peso en kilogramos. El algoritmo debe imprimir los datos leídos y la relación estatura-peso.

3. Elabore un algoritmo que lea dos datos enteros correspondientes a los catetos de un triángulo rectángulo y que calcule e imprima el valor de la hipotenusa de dicho triángulo.

4. Elabore un algoritmo que imprima los números enteros impares menores que 10.

5. Elabore un algoritmo que imprima la suma de los números enteros de 1 a 5.

6. Elabore un algoritmo que lea el código de un artículo, el precio unitario del artículo y la cantidad vendida. Su algoritmo debe calcular e imprimir el total de la venta, el IVA y el total a pagar, sabiendo que el impuesto es dieciséis por ciento.

7. Elabore un algoritmo que lea un entero de dos dígitos y produzca como salida los dígitos del número leído con su correspondiente mensaje. Por ejemplo, si el número leído es 27, la salida deberá ser:

   Primer dígito: 2
   Segundo dígito: 7

8. Elabore un algoritmo que lea dos números enteros y que produzca como salida la suma, resta, multiplicación, división y módulo del primero por el segundo.

9. Elabore un algoritmo que lea una temperatura en grados Fahrenheit y la convierta y la imprima en grados centígrados. Los grados Fahrenheit se convierten a grados centígrados restándoles 32 y multiplicando por cinco novenos.

10. Elabore un algoritmo que lea un dato, el cual es el valor de un ángulo en grados, y que lo convierta e imprima en radianes.

Capítulo 3 Estructura decisión

Contenido breve

Módulo 8
Estructura decisión
Módulo 9
Componente DE_LO_CONTRARIO
Módulo 10
Decisiones anidadas
Módulo 11
Selección múltiple
Módulo 12
Ejemplo de uso de la instrucción CASOS

En este capítulo se introducen las estructuras selectivas que se utilizan para controlar las acciones que se ejecutan de acuerdo al resultado de alguna condición. La instrucción SI y su componente opcional DE_LO_ CONTRARIO, y la instrucción CASOS, se describen como parte fundamental de un programa.

Las instrucciones de decisión ayudan a resolver importantes problemas de cálculo y de lógica en la elaboración de algoritmos.

Tomar decisiones es una actividad permanente en el diario vivir. En la elaboración de algoritmos es una herramienta fundamental.

Módulo 8

Estructura decisión

Introducción

Los algoritmos que se han elaborado hasta el momento se ejecutan usando únicamente la estructura secuencia, con sus instrucciones elementales: ASIGNACIÓN, LEA y ESCRIBA. Sin embargo, hay situaciones en las cuales es necesario no aplicar exactamente la estructura secuencia, es decir, no ejecutar las instrucciones exactamente en el orden en el cual se han escrito. Para lograr esta funcionalidad se cuenta con las instrucciones correspondientes a la estructura decisión.

Objetivos del módulo

1. Reconocer la estructura decisión.
2. Identificar cuándo y cómo utilizar decisiones.

Preguntas básicas

1. ¿En qué consiste la estructura decisión?
2. ¿Cuáles son las instrucciones correspondientes a la estructura decisión?
3. ¿Cuál es la forma general de la instrucción SI?

Contenidos del módulo

8.1 Definición de la estructura decisión
8.2 Instrucción SI, forma general
8.3 Ejemplo de algoritmo con la instrucción SI

«Tomar decisiones es una actividad cotidiana. Para ello debemos evaluar alguna condición. Por ejemplo: si estando en casa me invitan a una rumba, debo considerar si tengo dinero para salir; por lo tanto, evalúo lo siguiente: si tengo dinero, acepto la invitación; si no tengo dinero, permanezco en casa».

Vea en el botón **SI** del mapa conceptual el video "Módulo 8. Estructura decisión".

8.1 Definición de la estructura decisión

La estructura decisión permite instruir al computador para que ejecute ciertas acciones (instrucciones) según alguna condición.

Las instrucciones que pertenecen a la estructura decisión son:

1. La instrucción SI y su componente opcional DE_LO_CONTRARIO.
2. La instrucción CASOS.

8.2 Instrucción SI, forma general

La forma general de la instrucción SI es:

```
SI  condición
      Instrucciones que se ejecutan cuando la condición sea verdadera.
DE_LO_CONTRARIO
      Instrucciones que se ejecutan cuando la condición es falsa.
Fin(SI)
```

8.3 Ejemplo de algoritmo con la instrucción SI

Elabore un algoritmo que lea el salario actual de un empleado y que calcule e imprima el nuevo salario de acuerdo a la siguiente condición: si el salario es menor que 1000 pesos, aumentar el 10%; de lo contrario, no hacer aumento.

Análisis:

1. Datos de entrada:
   - Salario actual: salact

2. Datos de salida:
   - Aumento: au
   - Nuevo salario: nuesal

3. Cálculos:
   - Determinar el aumento según la condición planteada.

Nuestro algoritmo es:

```
1.    Algoritmo aumentoConCondicion(1)
2.          Variables:  salact, au, nuesal: numericaReal
3.          INICIO
4.                LEA(salact)
5.                SI  salact  <  1000
6.                      au = salact * 0.1
7.                DE_LO_CONTRARIO
8.                      au = 0
9.                Fin(SI)
10.               nuesal = salact + au
11.               ESCRIBA(salact, au, nuesal)
12          FIN
13.   Fin(aumentoConCondicion)
```

En la instrucción 2 se definen las variables con las cuales vamos a trabajar.

En la instrucción 4 se lee el salario actual.

En la instrucción 5 se compara el salario leído con el dato de referencia planteado en el enunciado.

Si la condición de la instrucción 5 es verdadera, se ejecuta la instrucción 6; de lo contrario se efectuará la instrucción 8.

En la instrucción 6 se determina el aumento, el cual es el diez por ciento del salario actual, mientras que en la instrucción 8 se asigna cero al aumento.

La instrucción 9 delimita el alcance de la instrucción SI.

En la instrucción 10 se calcula el nuevo salario, y en la instrucción 11 se escriben los datos leídos y los resultados pedidos.

El anterior algoritmo se puede escribir sin utilizar la componente DE_LO_CONTRARIO, la cual, como habíamos dicho, es opcional.

Veamos nuestro nuevo algoritmo:

```
1.    Algoritmo aumentoConCondicion(2)
2.        Variables:  salact, au, nuesal: numericaReal
3.        INICIO
4.            LEA(salact)
5.            au = 0
6.            SI  salact  <  1000
7.                au = salact * 0.1
8.            Fin(SI)
9.            nuesal = salact + au
10.           ESCRIBA(salact, au, nuesal)
11.       FIN
12.   Fin(aumentoConCondicion)
```

La diferencia de este segundo algoritmo con el primero es que al aumento inicialmente se le asigna el valor de cero en la instrucción 5.

Cuando se compara el salario actual con el valor de referencia (1000), se modificará el aumento sólo si el salario actual es menor que el valor de referencia; de lo contrario el aumento permanece en cero.

Resumen

En este módulo hemos presentado la primera parte de la instrucción de decisión SI, con un ejemplo de su utilización.

Ejercicios propuestos

1. Elabore un algoritmo que lea un número entero y que produzca el mensaje acerca de si el entero leído es par o impar.

2. Elabore un algoritmo que lea el nombre de una persona, su estatura y su peso. El algoritmo debe imprimir los datos leídos y un mensaje acerca de si la persona es obesa o no. Una persona se considera obesa si la relación estatura-peso es menor o igual que 2. Su programa debe leer la estatura en metros y el peso en kilogramos; además, la relación estatura-peso se calcula en centímetros sobre kilogramos.

3. Elabore un algoritmo que lea tres datos numéricos enteros y que determine si con esos tres datos se puede construir un triángulo equilátero. Su algoritmo debe imprimir: 'sí se puede' o 'no se puede'.

4. Elabore un algoritmo que lea año, mes y día de nacimiento de una persona, y año, mes y día actual. El algoritmo debe determinar e imprimir el mensaje apropiado si la edad de la persona está entre 18 y 25 años.

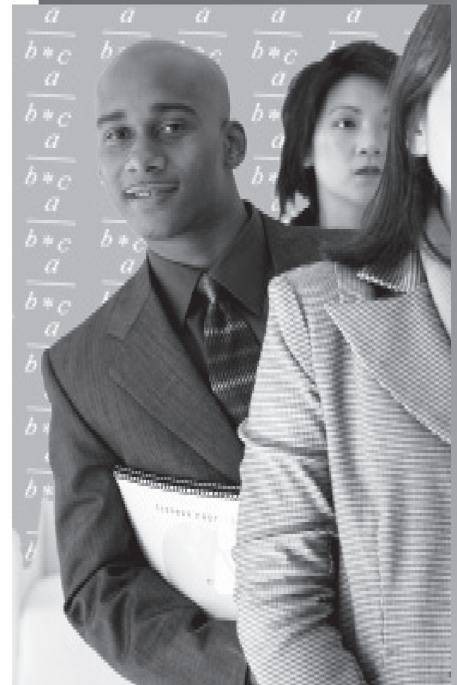
5. Elabore un algoritmo que lea dos números enteros, efectúe el producto y la suma de ellos, y luego imprima el resultado mayor entre la suma y el producto, con un mensaje apropiado.

6. Elabore un algoritmo que lea tres datos **a, b, c** correspondientes a la ecuación $ax^2 + bx + c$ y que produzca como salida las raíces de dicha ecuación.

7. Elabore un algoritmo que lea tres enteros positivos y que determine si pueden formar triángulo o no. Si pueden formar triángulo debe imprimir qué clase de triángulo es: equilátero, isósceles o escaleno. Tres enteros forman triángulo si cada uno de ellos es menor que la suma de los otros dos.

Módulo 9

Componente DE_LO_CONTRARIO



Introducción

En el módulo anterior se introdujo el uso de la instrucción condicional SI y se planteó que tiene una parte opcional que se denominó DE_LO_CONTRARIO. Cuando se elabora un algoritmo, es muy común que se presenten situaciones en las cuales las condiciones se pueden plantear así: utilizando la instrucción SI con su componente DE_LO_CONTRARIO, utilizando únicamente la instrucción SI sin su componente DE_LO_CONTRARIO o utilizando necesariamente la instrucción SI con su componente DE_LO_CONTRARIO.

En este módulo se tratarán ejemplos en los cuales es necesario utilizar la parte opcional DE_LO_CONTRARIO de la instrucción SI.

«En algunas situaciones la decisión a tomar es una elección entre dos opciones. Por ejemplo: tengo que ir a un baile, al cual debo llevar compañía. Si mi amiga favorita acepta mi invitación, voy con ella; de lo contrario, voy con la otra».

Objetivos del módulo

1. Reconocer cuándo es necesario el uso de la opción DE_LO_CONTRARIO.

Preguntas básicas

1. ¿Cuándo es necesario utilizar la componente DE_LO_CONTRARIO en una instrucción SI?
2. ¿Cuándo se puede obviar la componente DE_LO_CONTRARIO en una instrucción SI?

Contenidos del módulo

9.1 Ejemplo 1
9.2 Ejemplo 2

Vea en el botón **DE_LO_CONTRARIO** del mapa conceptual el video "Módulo 9. Opción DE_LO_CONRARIO".

9.1 Ejemplo 1

Elabore un algoritmo que lea un dato **d** y una característica, la cual puede ser 1 o 2. Si la característica leída es un 1, el dato leído corresponde al radio de un círculo; si la característica leída es un 2, el dato leído corresponde al lado de un cuadrado. El algoritmo debe calcular e imprimir el área del círculo o del cuadrado, con su correspondiente mensaje: círculo o cuadrado.

Análisis:

1. Datos de entrada:
   - Característica: c
   - Dato: d

2. Datos de salida:
   - Área de la figura: area

3. Cálculos:
   - Determinar el área de la figura según la característica leída.

Nuestro algoritmo es:

```
1.      Algoritmo areaFiguraConMensaje
2.          Variables: d: numericaEntera
3.                     area: numericaReal
4.          INICIO
5.              LEA(c, d)
6.              SI  c == 1
7.                  area = 3.14 * d ^ 2
8.                  ESCRIBA("área del círculo: ", area)
9.              DE_LO_CONTRARIO
10.                 area = d ^ 2
11.                 ESCRIBA("área del cuadrado: ", area)
12.             Fin(SI)
13.         FIN
14.     Fin(areaFiguraConMensaje)
```

En las instrucciones 2 y 3 definimos las variables con las cuales trabajará el algoritmo.

En la instrucción 5 se leen los datos de trabajo.

En la instrucción 6 comparamos la característica leída con 1. Si la característica es 1 se ejecutan las instrucciones 7 y 8; si no, se ejecutan las instrucciones 10 y 11.

Si los datos leídos son 1 y 10, entonces **c** queda valiendo 1 y **d** queda valiendo 10. Al ejecutar la instrucción 6 la condición es verdadera y se ejecutan las instrucciones 7 y 8, es decir, la variable **area** queda valiendo 314.0 e imprime:

área del círculo: 314

Si los datos leídos hubieran sido 2 y 10, entonces **c** queda valiendo 2 y **d** queda valiendo 10. Al ejecutar la instrucción 6 la condición es falsa y por lo tanto ejecuta las instrucciones correspondientes a la parte DE_LO_CONTRARIO, o sea que la variable **area** queda valiendo 100 e imprime:

área del cuadrado: 100

Si no hubiéramos tenido que producir el mensaje de si la figura es un círculo o un cuadrado, la componente DE_LO_CONTRARIO se puede evitar.

Veamos cómo quedaría nuestro algoritmo:

```
1.    Algoritmo areaFiguraSinMensaje
2.        Variables: d: numericaEntera
3.                   area: numericaReal
4.        INICIO
5.            LEA(c, d)
6.            area = d ^ 2
7.            SI  c == 1
8.                area = 3.14 * area
9.            Fin(SI)
10.           ESCRIBA(area)
11.       FIN
12.   Fin (areaFiguraSinMensaje)
```

Como podrá observar, en el algoritmo *areaFiguraConMensaje* el cálculo del área del cuadrado y del círculo tienen en común que hay que elevar el dato **d** al cuadrado. Si se trata de un círculo, este resultado hay que multiplicarlo por 3.14; de lo contrario no hay que efectuar más operaciones. Usando esta propiedad escribimos la instrucción 6, y luego en la instrucción 7 averiguamos si **c** es igual a 1; en caso de serlo modificamos el contenido de la variable **area**; si no, el contenido de la variable **area** permanece intacto.

Observe que en este segundo algoritmo el usuario no sabrá si el dato impreso (área) es un círculo o un cuadrado.

9.2 Ejemplo 2

Elabore un algoritmo que lea dos datos numéricos enteros y que los imprima ordenados ascendentemente.

Análisis:

1. Datos de entrada:
   - Dos datos enteros: a, b

2. Datos de salida:
   - Los mismos de entrada

3. Cálculos:
   - Identificar cuál es mayor para imprimirlo de primero.

```
1.    Algoritmo dosDatosOrdenados
2.        Variables: a, b: numericaEntera
3.        INICIO
4.            LEA(a, b)
5.            SI  a < b
```

```
6.              ESCRIBA(a, b)
7.          DE_LO_CONTRARIO
8.              ESCRIBA(b, a)
9.          Fin(SI)
10.     FIN
11. Fin(dosDatosOrdenados)
```

En la instrucción 2 se definen las variables con las cuales vamos a trabajar.

En la instrucción 4 se leen dichos datos.

En la instrucción 5 se comparan los datos leídos. Si el contenido de la variable **a** es menor que el contenido de la **b**, ejecuta la instrucción 6, es decir, escribe primero el contenido de la variable **a** y luego el contenido de la **b;** si no, ejecuta la instrucción 8, es decir, escribe primero el contenido de la variable **b** y luego el contenido de la **a**.

Con base en lo anterior, si los datos entrados son 3 y 6 nuestro algoritmo escribe

3 6

Si los datos entrados hubieran sido 8 y 2 nuestro algoritmo escribe

2 8

Resumen

En este módulo presentamos la segunda parte de la instrucción de decisión SI, con dos ejemplos en los cuales se muestra su aplicación.

Ejercicios propuestos

1. Elabore un algoritmo que lea un número entero y que determine e imprima el mensaje apropiado si el número leído es impar y menor que 100.

2. Elabore un algoritmo que lea un número entero y que produzca como salida el número leído con el mensaje ‘es positivo’ o ‘es negativo’ según el caso.

3. Elabore un algoritmo que lea el nombre de un estudiante y las tres notas obtenidas en los exámenes de una materia. El algoritmo debe calcular la definitiva y producir un mensaje de felicitación si la materia fue aprobada, o un mensaje de reproche si la materia fue reprobada. La materia se aprueba si obtiene como definitiva una nota mayor o igual que 3.0.

4. Elabore un algoritmo que lea dos números enteros de dos dígitos y determine si tienen algún dígito en común.

5. Elabore un algoritmo que lea dos números enteros de tres dígitos y que determine e imprima el mensaje apropiado si ambos números terminan en el mismo dígito.

Módulo 10

Decisiones anidadas

Introducción

En los módulos anteriores se ha tratado la instrucción SI, con su componente opcional DE_LO_CONTRARIO, de una manera simple. En la práctica se presentan hechos en los cuales es necesario controlar situaciones dentro de situaciones ya controladas, es decir, comprobar condiciones dentro de condiciones, o comprobar varias condiciones a la vez. Estos acontecimientos implican el uso de la instrucción SI de una forma más compleja, la cual se tratará en este módulo.

Objetivos del módulo

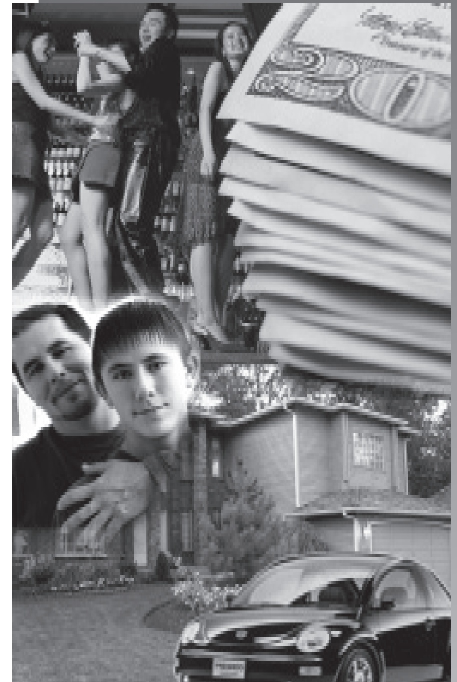
1. Elaborar algoritmos más complejos utilizando la instrucción SI con su componente opcional DE_LO_CONTRARIO.

Preguntas básicas

1. Qué es mejor: ¿condición compuesta o varias instrucciones SI anidadas?
2. ¿Cómo afecta la ejecución de un algoritmo el uso, o no, de la opción DE_LO_CONTRARIO?

Contenidos del módulo

10.1 Ejemplo de ilustración



«Algunas veces la toma de una decisión es más compleja que una simple elección entre dos opciones. La invitación puede ser a una finca, a un cine o a un baile. Para ir a la finca o al cine necesito automóvil, a la finca 50.000 pesos, al cine 20.000, y al baile ni automóvil ni dinero. Para tomar una decisión evalúo lo siguiente: si mi padre me presta el carro y mi capital es mayor que 50.000 pesos, voy a la finca; si mi padre me presta el carro y mi capital es mayor que 20.000 pesos, voy al cine; si mi padre no me presta el carro, voy al baile».

Vea en el botón **DE_LO_CONTRARIO** del mapa conceptual el video "Módulo 10. Instrucción Si".

10.1 Ejemplo de ilustración

Elabore un algoritmo que lea tres datos numéricos enteros y que los imprima ordenados ascendentemente.

Análisis:

1. Datos de entrada:
   - Tres datos numéricos enteros: a, b, c

2. Datos de salida:
   - Los mismos tres datos de entrada ordenados ascendentemente.

3. Cálculos:
   - Determinar el menor de los tres datos para imprimirlo de primero, luego determinar el menor de los otros dos para imprimirlo de segundo y luego imprimir el tercer dato.

Una primera forma de escribir este algoritmo es siendo exhaustivos en la comparación de los datos.

Realmente, las diferentes situaciones que se pueden presentar para escribir los tres datos son:

La primera, cuando **a** es menor que **b** y **b** es menor que **c**.
La segunda, cuando **a** es menor que **c** y **c** es menor que **b**.
La tercera, cuando **b** es menor que **a** y **a** es menor que **c**,
y así sucesivamente.

A cada situación le corresponde una relación de orden diferente.

1. a, b, c
2. a, c, b
3. b, a, c
4. b, c, a
5. c, a, b
6. c, b, a

Un algoritmo para efectuar esta tarea es:

```
1.     Algoritmo ordenaTresDatos(1)
2          Variables:  a, b, c:  numericaEntera
3.         INICIO
4.             LEA(a, b, c)
5.             SI a <= b && b <= c
6.                 ESCRIBA(a, b, c)
7.             Fin(SI)
8.             SI a <= c && c <= b
9.                 ESCRIBA(a, c, b)
10.            Fin(SI)
11.            SI b <= a && a <= c
12.                ESCRIBA(b, a, c)
13.            Fin(SI)
```

```
14.                SI b <= c && c <= a
15.                     ESCRIBA(b, c, a)
16.                Fin(SI)
17.                SI c <= a  &&  a <= b
18.                     ESCRIBA(c, a, b)
19.                Fin(SI)
20.                SI c <= b &&  b <= a
21.                     ESCRIBA(c, b, a)
22.                Fin(SI)
23.           FIN
24.     Fin(ordenaTresDatos)
```

Las instrucciones 5 a 7 consideran la primera situación; las instrucciones 8 a 10 consideran la segunda; las instrucciones 11 a 13 consideran la tercera; las instrucciones 14 a 16 consideran la cuarta; las instrucciones 17 a 19 consideran la quinta; y las instrucciones 20 a 22 consideran la sexta.

Este algoritmo tiene el inconveniente de que cuando una situación sea verdadera, continúa preguntando por las demás situaciones, lo cual genera ineficiencia. Para evitar esta ineficiencia utilizamos la parte opcional DE_LO_CONTRARIO.

Veamos cómo queda nuestro algoritmo:

```
1.     Algoritmo ordenaTresDatos(2)
2          Variables:  a, b, c:  numericaEntera
3.         INICIO
4.              LEA(a, b, c)
5.              SI a <= b && b <= c
6.                   ESCRIBA(a, b, c)
7.              DE_LO_CONTRARIO
8.                   SI a <= c && c <= b
9.                        ESCRIBA(a, c, b)
10.                  DE_LO_CONTRARIO
11.                       SI b <= a && a <= c
12.                            ESCRIBA(b, a, c)
13.                       DE_LO_CONTRARIO
14.                            SI b <= c && c <= a
15.                                 ESCRIBA(b, c, a)
16.                            DE_LO_CONTRARIO
17.                                 SI c <= a  &&  a <= b
18.                                     ESCRIBA(c, a, b)
19.                                 DE_LO_CONTRARIO
20.                                     ESCRIBA(c, b, a)
21.                                 Fin(SI)
22.                            Fin(SI)
23.                       Fin(SI)
24.                  Fin(SI)
25.             Fin(SI)
26.        FIN
27.    Fin(ordenaTresDatos)
```

De esta manera, cuando encuentre que una condición (situación) es verdadera, procede a imprimir los datos en forma ordenada y no sigue preguntando por las demás condiciones.

Una tercera forma en que podemos elaborar el algoritmo es la siguiente:

Comparamos **a** con **b**.

Pueden suceder dos cosas: una, que **a** sea menor que **b,** y dos, que **b** sea menor que **a**.

1. Si **a** es menor que **b**, implica que habrá que escribir el dato **a** antes que el dato **b**; por lo tanto, las posibles formas de escribir los tres datos son:

   1. **a**, **b**, c
   2. **a**, c, **b**
   3. c, **a**, **b**

   Si **b** es menor que **c,** imprimimos la primera posibilidad: a, b, c; de lo contrario, debemos comparar **a** con **c**.

   Si **a** es menor que **c,** imprimimos la segunda posibilidad: a, c, b; de lo contrario, imprimimos la tercera posibilidad: c, a, b.

2. Si **b** es menor que **a** implica que habrá que escribir el dato **b** antes que el dato **a**; por tanto las posibles formas de escribir los tres datos son:

   1. b, a, c
   2. b, c, a
   3. c, **b**, **a**

   Si **a** es menor que **c,** imprimimos la primera posibilidad: b, a, c; de lo contrario, debemos comparar **b** con **c**.

   Si **b** es menor que **c,** imprimimos la segunda posibilidad: b, c, a; de lo contrario, imprimimos la tercera posibilidad: c, b, a.

Con base en el anterior análisis nuestro algoritmo queda:

```
1.    Algoritmo ordenaTresDatos(3)
2.        Variables: a, b, c: numericaEntera
3.        INICIO
4.            LEA(a, b, c)
5.            SI  a <= b
6.                SI  b <= c
7.                    ESCRIBA(a, b ,c)
8.                DE_LO_CONTRARIO          // c es menor que b
9.                    SI  a <= c
10.                       ESCRIBA(a, c, b)
11.                   DE_LO_CONTRARIO   // c es menor que a
12.                       ESCRIBA(c, a, b)
13.                   Fin(SI)
14.               Fin(SI)
15.                DE_LO_CONTRARIO          // b es menor que a
16.               SI  a <= c
17.                   ESCRIBA(b, a, c)
18.               DE_LO_CONTRARIO          // c es menor que a
19.                   SI  b <= c
```

```
20.                                    ESCRIBA(b, c, a)
21.                               DE_LO_CONTRARIO    // c es menor que b
22.                                    ESCRIBA(c, b, a)
23.                               Fin(SI)
24.                         Fin(SI)
25.                   Fin(SI)
26.              FIN
27.      Fin(ordenaTresDatos)
```

Veamos cómo se comportaría este último algoritmo con algunos datos de entrada:

	a	b	c
1.º	3	1	6
2.º	5	6	2
3.º	7	6	5
4.º	1	2	3

Al ejecutar la instrucción de lectura con los datos de la primera línea, los datos almacenados en las variables **a**, **b** y **c** son 3, 1 y 6, respectivamente.

El resultado de ejecutar la instrucción 5 (comparar **a** con **b**) es falso; por lo tanto la ejecución continúa en la instrucción 16, en la cual compara **a** con **c**, obteniendo un resultado verdadero y, como consecuencia, ejecuta la instrucción 17, en la cual escribe **b**, **a**, **c**, lo cual es 1, 3, 6.

Si se leyeran los datos de la segunda línea, los datos almacenados en las variables **a**, **b** y **c** son 5, 6 y 2.

El resultado de ejecutar la instrucción 5 es verdad; por lo tanto, continúa ejecutando la instrucción 6 (compara **b** con **c**), la cual produce un resultado falso, lo que implica que la siguiente instrucción que se ejecuta es la instrucción 9 (compara **a** con **c**), la cual produce un resultado falso y por consiguiente ejecutará la instrucción 12: escribe **c**, **a**, **b**: 2, 5, 6.

Dejamos, como ejercicio, que el estudiante analice el comportamiento del algoritmo con los datos de las líneas tercera y cuarta.

Resumen

En este módulo hemos tratado cómo se pueden elaborar algoritmos más complejos utilizando la instrucción SI en forma anidada, es decir, instrucciones de decisión dentro de instrucciones de decisión.

Ejercicios propuestos

1. Elabore un algoritmo que lea un número entero menor que 32768 y que determine cuántos dígitos tiene.

2. Elabore un algoritmo que lea un número entero menor que 32768 y que determine si el número leído es múltiplo de la suma de sus dígitos.

3. Elabore un algoritmo que lea un número entero menor que 10 y determine si es primo o no.

4. Elabore un algoritmo que valide que un número entero leído tenga tres dígitos y que luego determine en cuál posición se halla el mayor de los tres dígitos.

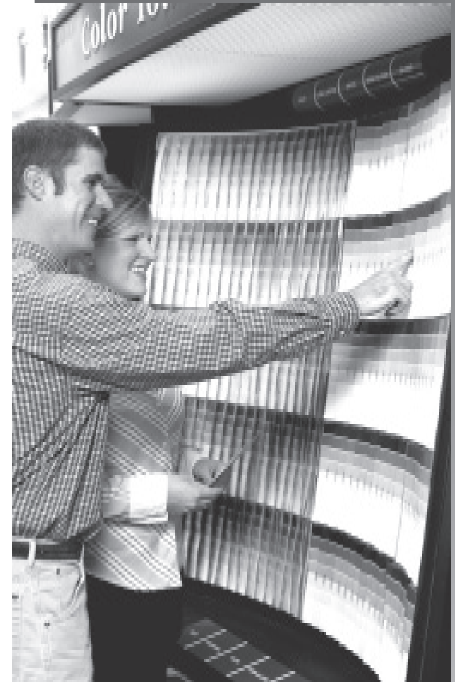
5. Elabore un algoritmo que lea tres números enteros y que sólo imprima el mayor de ellos.

6. Elabore un algoritmo que lea un número de dos dígitos y determine si alguno de ellos es impar.

7. Elabore un algoritmo que lea un número entero de cinco dígitos menor que 32768 y determine si sus dígitos constituyen un palíndromo. Palíndromo es una hilera que se lee igual de izquierda a derecha que de derecha a izquierda.

Módulo 11

Selección múltiple



Introducción

Ya hemos visto que la estructura decisión se puede construir con la instrucción SI y su componente opcional DE_LO_CONTRARIO. Sin embargo, hay situaciones en las cuales su utilización genera algoritmos poco legibles y quizás más complejos. Con el fin de evitar estos inconvenientes se ha creado la instrucción CASOS, la cual permite construir algoritmos más legibles y menos complejos. En este módulo se trata dicha instrucción.

Objetivos del módulo

1. Reconocer y aplicar la instrucción CASOS.

Preguntas básicas

1. ¿En qué consiste la instrucción CASOS?
2. ¿Cuándo se aplica la instrucción CASOS?
3. ¿En qué consiste la opción SALTE?

Contenidos del módulo

11.1 Forma general
11.2 Ejemplo de uso

«En ocasiones la toma de una decisión implica acciones diferentes por efectuar. Por ejemplo: de acuerdo al sitio que elija para ir de vacaciones será el equipaje que debo llevar. Las opciones son: ir a la playa, ir de pesca al lago, ir de camping *a un sitio de clima medio o ir a esquiar a las montañas».*

Vea en los botones **CASOS** y **SALTE** del mapa conceptual el video "Módulo 11. Instrucción CASOS".

11.1 Forma general

La forma general de la instrucción CASOS es:

```
CASOS
    :condición 1: instrucciones que se ejecutan cuando la condición 1 es verdadera.
          SALTE
    :condición 2: instrucciones que se ejecutan cuando la condición 2 es verdadera.
          SALTE
    .
    .
    .
    :condición n: instrucciones que se ejecutan cuando la condición n es verdadera.
          SALTE

    :OTRO_CASO: instrucciones que se ejecutan cuando ninguna de las condiciones anteriores es verdadera.

Fin(CASOS)
```

La componente SALTE presente al final de cada condición es opcional, y veremos su funcionalidad más adelante, con los ejemplos.

11.2 Ejemplo de uso

Elabore un algoritmo que lea el nombre de una persona y su estado civil.

El estado civil está codificado con un dígito con los siguientes significados:

1: Soltero
2: Casado
3: Separado
4: Viudo
5: Unión libre

El algoritmo debe imprimir el nombre leído y la descripción correspondiente al estado civil leído.

Análisis:

1. Datos de entrada:
    - Nombre de la persona: nom
    - Estado civil: ec

2. Datos de salida:
    - Nombre de la persona: nom
    - Estado civil: ec

3. Cálculos:
    - Comparar el estado civil leído según el código establecido e imprimir la descripción correspondiente.

Nuestro algoritmo es:

```
1.    Algoritmo estadoCivil(1)
2.        Variables: nom: alfanumérica
3.                   ec: numericaEntera
4.        INICIO
5.            LEA(nom, ec)
6.            CASOS
7.                :ec == 1: ESCRIBA(nom, " soltero")
8.                :ec == 2: ESCRIBA(nom, " casado")
9.                :ec == 3: ESCRIBA(nom, " separado")
10.               :ec == 4: ESCRIBA(nom, " viudo")
11.               :ec == 5: ESCRIBA(nom, " unión libre")
12.           :OTRO_CASO: ESCRIBA(ec, " estado civil inválido")
13.           Fin(CASOS)
14.       FIN
15.   Fin(estadoCivil)
```

Dentro de la instrucción CASOS cuando escribimos **:ec == 1:** la máquina compara **ec** con 1; si **ec** es igual a 1 ejecuta las instrucciones correspondientes a ese caso.

Un algoritmo equivalente al anterior, utilizando la instrucción SI, es el siguiente:

```
1.    Algoritmo estadoCivil(2)
2.        Variables: nom: alfanumérica
3.                   ec: numericaEntera
4.        INICIO
5.            LEA(nom, ec)
6.            SI  ec == 1
7.                ESCRIBA(nom, " soltero")
8.            Fin(SI)
9.            SI  ec == 2
10.               ESCRIBA(nom, " casado")
11.           Fin(SI)
12.           SI  ec == 3
13.               ESCRIBA(nom, " separado")
14.           Fin(SI)
15.           SI  ec == 4
16.               ESCRIBA(nom, " viudo")
17.           Fin(SI)
18.           SI  ec == 5
19.               ESCRIBA(nom, " unión libre")
20.           Fin(SI)
21.           SI ec ≠ 1 && ec ≠ 2 && ec ≠ 3 && ec ≠ 4 && ec ≠ 5
22.               ESCRIBA(ec, " estado civil inválido")
23.           Fin(SI)
24.       FIN
25.   Fin(estadoCivil)
```

Este algoritmo tiene el “inconveniente” de que si el estado civil es 1, al hacer la primera comparación escribe el nombre y la descripción “soltero” y sigue preguntando inoficiosamente si el estado civil es 2, o 3, o 4, o 5, y más aún, en la instrucción 21 pregunta si el estado civil es diferente de 1.

Una mejor forma de hacerlo es evitando preguntas inoficiosas, para lo cual utilizamos la opción DE_LO_CONTRARIO.

Nuestro algoritmo quedará así:

```
1.   Algoritmo estadoCivil(3)
2.       Variables nom: alfanumérica
3.                       ec: numericaEntera
4.       INICIO
5.           LEA(nom, ec)
6.           SI  ec == 1
7.               ESCRIBA(nom, " soltero")
8.           DE_LO_CONTRARIO
9.               SI  ec == 2
10.                  ESCRIBA(nom, " casado")
11.              DE_LO_CONTRARIO
12.                  SI  ec == 3
13.                      ESCRIBA(nom, " separado")
14.                  DE_LO_CONTRARIO
15.                      SI  ec == 4
16.                          ESCRIBA(nom, " viudo")
17.                      DE_LO_CONTRARIO
18.                          SI  ec == 5
19.                              ESCRIBA(nom, " unión libre")
20.                          DE_LO_CONTRARIO
21                               ESCRIBA(ec, " estado civil inválido")
22.                          Fin(SI)
23.                      Fin(SI)
24.                  Fin(SI)
25.              Fin(SI)
26.          Fin(SI)
27.      FIN
28.  FIN(estadoCivil)
```

Ya con este algoritmo evitamos hacer preguntas inoficiosas.

Si utilizamos la instrucción CASOS y queremos evitar las preguntas inoficiosas, usamos la componente opcional SALTE. Cuando el grupo de instrucciones correspondiente a una condición termina con la instrucción SALTE, se está instruyendo a la máquina para que no siga averiguando por las demás condiciones y que continúe la ejecución en la instrucción siguiente al fin de los CASOS.

Nuestro algoritmo *estadoCivil*, utilizando la instrucción CASOS con la componente SALTE, queda así:

```
1.      Algoritmo estadoCivil(1)
2.          Variables: nom: alfanumérico
3.                      ec: numericaEntera
4.          INICIO
5.              LEA(nom, ec)
6.              CASOS
7.                  :ec == 1: ESCRIBA(nom, " soltero")
8.                            SALTE
```

```
9.                         :ec == 2: ESCRIBA(nom,  " casado")
10.                                  SALTE
11.                        :ec == 3: ESCRIBA(nom, " separado")
12.                                  SALTE
13.                        :ec == 4: ESCRIBA(nom, " viudo")
14.                                  SALTE
15.                        :ec == 5: ESCRIBA(nom, " unión libre")
16.                                  SALTE
17.                        :OTRO_CASO: ESCRIBA(ec, " estado civil inválido")
18.                  Fin(CASOS)
19.           FIN
20.     Fin(estadoCivil)
```

Un punto importante que se debe considerar en este sitio es: ¿cuándo utilizar la instrucción SI, y cuándo utilizar la instrucción CASOS?

La respuesta es muy sencilla:

- Cuando el resultado de una comparación sólo da dos alternativas se utiliza la instrucción SI.
- Cuando el resultado de una comparación da más de dos alternativas se utiliza la instrucción CASOS.

Es conveniente también hacer notar que no siempre es recomendable utilizar la componente SALTE.

Resumen

En este módulo hemos presentado la instrucción de decisión CASOS, que hace más fácil evitar muchas instrucciones SI y logra que el algoritmo sea más sencillo y legible.

Ejercicios propuestos

1. Elabore un algoritmo que lea año y mes y que imprima el número de días que tiene el mes leído.

2. Elabore un algoritmo que lea un número entero y que imprima el número de dígitos que tiene el número. El máximo número entero permitido es 32767.

3. Elabore un algoritmo que lea el nombre de una persona y su edad en años. El algoritmo debe imprimir el nombre de la persona y su clasificación de acuerdo a los siguientes criterios: la persona es 'niño', si la edad está entre 0 y 10 años; 'adolescente', si la edad está entre 11 y 18 años; 'adulto', si la edad está entre 19 y 35 años; 'maduro', si la edad está entre 36 y 60 años; y 'anciano', si la edad es mayor de 60 años.

4. Elabore un algoritmo que lea el nombre de un estudiante y su promedio acumulado de la carrera. El algoritmo debe imprimir el nombre del estudiante y alguno de los siguientes mensajes: 'pésimo', 'malo', 'regular', 'bueno' o 'excelente'. El estudiante se considera pésimo si el promedio acumulado es menor o igual que 1; malo, si el promedio es mayor que 1 y menor que 3; regular, si el promedio es mayor o igual que 3 y menor que 4; bueno, si el promedio es mayor o igual que 4 y menor que 4.5; y excelente, si el promedio es mayor o igual que 4.5.

5. Elabore un algoritmo que lea un número entero menor que 32768 y que efectúe lo siguiente: si es múltiplo de 4, imprimir el número dividido por 4; si es múltiplo de 5, imprimir la quinta parte del número elevada al cuadrado; si es múltiplo de 7, imprimir el número dividido por 8; y si no es múltiplo de ninguno de los anteriores, producir el mensaje ‘número extraño’.

Módulo 12

Ejemplos de uso de la instrucción CASOS

Introducción

En los módulos anteriores se han presentado los conceptos básicos elementales para elaborar algoritmos, junto con las instrucciones correspondientes a la estructura decisión. En este módulo se presenta un ejemplo en el cual se aplican conjuntamente los conceptos tratados hasta ahora, con las instrucciones de la estructura decisión.

Objetivos del módulo

1. Aplicar los temas tratados hasta ahora en la elaboración de un algoritmo.

Preguntas básicas

1. ¿Cuándo se utiliza la instrucción CASOS y cuándo la instrucción SI?
2. ¿Cuándo se utiliza la opción SALTE?

Contenidos del módulo

12.1 Ejemplo de uso

«En la práctica se presentan situaciones que comprenden todas las acciones que se enunciaron en la presentación del módulo 11: ir a la playa, ir de pesca al lago, ir de camping *a un sitio de clima medio o ir a esquiar a las montañas. Si elijo ir a la playa debo verificar que los vestidos de baño todavía me sirvan, que tenga bronceador, que tenga lentes oscuros, en fin, una infinidad de factores; si voy de pesca al lago debo verificar que mis aparejos de pesca se encuentren en buen estado, debo llevar un libro para leer, debo comprar carnadas, etc. En resumen, cualquiera que sea la elección debo controlar otro montón de circunstancias».*

Vea en los botones **CASOS** y **SALTE** del mapa conceptual el video "Módulo 12. Ejercicio de aplicación".

12.1 Ejemplo de uso

Elabore un algoritmo que lea: nombre de un empleado, estado civil, edad y salario actual. Para el empleado leído determine el nuevo salario con base en las siguientes políticas:

- Soltero menor de 30 años se le aumenta el 10% de su salario actual.
- Soltero mayor de 30 años se le aumenta el 12% de su salario actual.
- Casado menor de 25 años se le aumenta el 12% de su salario actual.
- Casado mayor de 25 años se le aumenta el 15% de su salario actual.
- Separado menor de 20 años se le aumenta el 8% de su salario actual.
- Separado mayor de 20 años se le aumenta el 10% de su salario actual.
- Viudo menor de 30 años se le aumenta el 15% de su salario actual.
- Viudo mayor de 30 años se le aumenta el 12% de su salario actual.
- Empleado en unión libre que devengue menos de 1000 pesos se le aumenta el 20% de su salario actual.
- Empleado en unión libre que devengue 1000 pesos o más se le aumenta el 12% de su salario actual.

El algoritmo deberá determinar el aumento del empleado leído e imprimir: nombre, estado civil, descripción del estado civil, edad, salario actual, porcentaje de aumento, aumento y nuevo salario. Para imprimir la descripción del estado civil usaremos la convención definida en nuestro algoritmo *estadoCivil* del módulo anterior.

Análisis:

1. Datos de entrada:
   - Nombre del empleado: nom
   - Estado civil: ec
   - Edad: ed
   - Salario actual: sa

2. Datos de salida:
   - Nombre del empleado: nom
   - Estado civil: ec
   - Descripción del estado civil: dec
   - Edad: ed
   - Salario actual: sa
   - Porcentaje de aumento: pau
   - Aumento: au
   - Nuevo salario: ns

3. Cálculos:
   - Con base en las políticas de la empresa se debe calcular el nuevo salario del empleado. El nuevo salario se calcula sumándole al salario actual el aumento, el cual se obtiene multiplicando el salario actual por el porcentaje de aumento que le corresponde al empleado.

Nuestro algoritmo queda:

```
1.    Algoritmo salarios
2.        Variables : nom, dec: alfanuméricas
3.                    ec, ed: numericaEntera
4.                    sa, pau, au, ns: numericaReal
5.        INICIO
6.            LEA(nom, ec, ed, sa)
```

```
7.          CASOS
8.             :ec == 1 : dec = “soltero”
9.                      SI  ed < 30
10.                        pau = 0.10
11.                     DE_LO_CONTRARIO
12.                        pau = 0.12
13.                     Fin(SI)
14.                     SALTE
15.            :ec == 2 : dec = “casado”
16.                     SI  ed < 25
17.                        pau = 0.12
18.                     DE_LO_CONTRARIO
19.                        pau = 0.15
20.                     Fin(SI)
21.                     SALTE
22.            :ec == 3 : dec = “separado”
23.                     SI  ed < 20
24.                        pau = 0.08
25.                     DE_LO_CONTRARIO
26.                        pau = 0.10
27.                     Fin(SI)
28.                     SALTE
29.            :ec == 4 : dec = “viudo”
30.                     SI  ed < 30
31.                        pau = 0.15
32.                     DE_LO_CONTRARIO
33.                        pau = 0.12
34.                     Fin(SI)
35.                     SALTE
36.            :ec == 5 : dec = “unión libre”
37.                     SI  sa < 1000
38.                        pau = 0.20
39.                     DE_LO_CONTRARIO
40.                        pau = 0.12
41.                     Fin(SI)
42.                     SALTE
43.            :OTRO_CASO :   dec = “estado civil inválido”
44.                           pau = 0.0
45.         Fin(CASOS)
46.         au = sa * pau
47.         ns = sa + au
48.         ESCRIBA(nom, ec, dec, ed, sa, pau, au, ns)
49.      FIN
50.   Fin(salarios)
```

El cuerpo general de nuestro algoritmo está constituido por la instrucción CASOS de la estructura decisión, teniendo como variable principal el estado civil de la persona.

Para cada estado civil actualizamos el campo **dec**, en el cual almacenamos la descripción del estado civil correspondiente al código de estado civil leído. Además, determinamos el porcentaje de aumento correspondiente, según las políticas de la empresa: unas de acuerdo a la edad y otras de acuerdo al salario actual.

Teniendo definido el porcentaje de aumento del empleado, procedemos a determinar su aumento, su nuevo salario y a imprimir los datos pedidos, lo cual se hace con las instrucciones 46 a 48.

Resumen

En este módulo presentamos un ejemplo completo de aplicación de las instrucciones de decisión: la instrucción SI combinada con la instrucción CASOS.

Ejercicios propuestos

1. Haga prueba de escritorio al algoritmo de este módulo para cada uno de los siguientes conjuntos de datos:

nom	ec	ed	sa
Juan	1	28	2500
Luis	2	31	800
Ana	3	18	1600
Luz	4	45	3000
Clara	5	33	2500
Inés	3	25	1000
Lulú	1	39	700

2. Elabore un algoritmo que lea los siguientes datos: nombre de una muchacha, edad, estatura, peso y estado civil. El algoritmo debe clasificar cada chica de acuerdo a los siguientes criterios: si es menor de 15 años y mide menos de 160 cm es una mafalda; si está entre 18 y 25 años, mide más de 175 cm y pesa menos de 65 kg es una barbie; si es mayor de 50 años, mide más de 168 cm, pesa más de 80 kg y es casada es una helga; si es casada, mayor de 25 años y menor de 30 años y pesa menos de 70 kg es una reina; y si es soltera, mide más de 180 cm, pesa entre 65 y 75 kg y tiene menos de 21 años es una princesa.

3. La empresa de productos de belleza ‘El acné’ otorga descuento a sus clientes según la siguiente clasificación: si es mayorista, tiene una antigüedad de más de dos años y el valor de la compra es mayor que 2.000.000 de pesos le da un descuento del 25%; si es mayorista, tiene una antigüedad menor o igual que dos años y el valor de la compra está entre 1.500.000 y 2.000.000 de pesos le da un descuento de 20%; si es minoritario, tiene una antigüedad superior a cinco años y el valor de la compra es superior 2.000.000 de pesos le da un descuento del 18%; si es ocasional y el valor de la compra es superior a 2.000.000 de pesos le da un descuento del 10%. En cualquier otro caso, la compañía no da ningún descuento. Elabore un algoritmo que lea la clase de cliente, la antigüedad y el valor de compra y determine el valor a pagar por la compra.

   Los códigos de clasificación del cliente son:

   1. Mayorista
   2. Minorista
   3. Ocasional

Capítulo 4 — Estructura ciclo

Contenido breve

Módulo 13
Estructura ciclo e instrucción MIENTRAS
Módulo 14
Contadores y acumuladores
Módulo 15
Aplicaciones de ciclos en matemáticas
Módulo 16
Ciclos anidados con instrucción MIENTRAS
Módulo 17
Instrucción PARA
Módulo 18
Ciclos anidados con instrucción PARA
Módulo 19
Instrucción HAGA

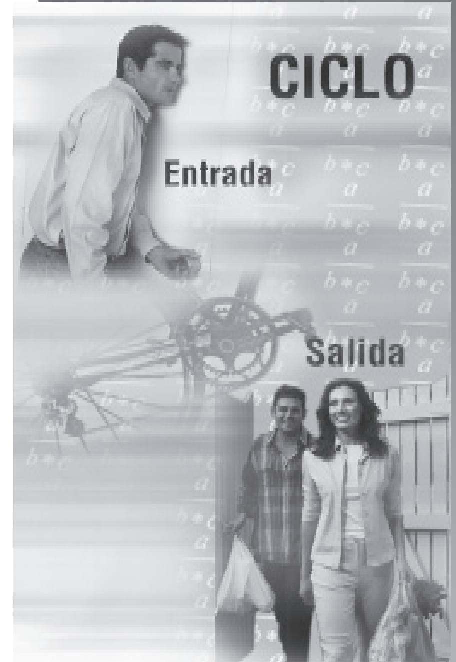
Los algoritmos elaborados hasta este punto han tratado temas de programación como entradas, salidas, asignaciones, expresiones, secuencias y decisiones. Sin embargo, hay muchas situaciones en las cuales se requiere que algunas operaciones o secuencia de instrucciones se repitan una y otra vez utilizando diferentes datos.

En este capítulo se estudian las diferentes instrucciones que se utilizan para instruir al computador con el fin de que ejecute más de una vez cierto conjunto de instrucciones de acuerdo con alguna condición establecida.

Ejecutar la misma actividad varias veces, con diferentes datos, clientes, etc., es una situación común en la vida diaria.

Módulo 13

Estructura ciclo e instrucción MIENTRAS



Introducción

En los ejercicios desarrollados anteriormente leíamos el salario actual de un empleado y determinábamos el aumento con base en un porcentaje leído o en una condición específica (que gane menos de mil pesos, por ejemplo).

Realmente, cuando se desea actualizar salarios, no es para un solo empleado, sino para todos los empleados de la empresa. Con el algoritmo que habíamos desarrollado, si deseamos actualizar el salario de todos los empleados debemos ejecutarlo tantas veces como empleados tenga la empresa, lo cual es una tarea muy dispendiosa. Para evitar esto podemos escribir un algoritmo con el cual procesemos todos los empleados utilizando una instrucción de ciclo.

Objetivos del módulo

1. Reconocer la estructura ciclo y sus instrucciones.
2. Reconocer la instrucción MIENTRAS.
3. Identificar el concepto de contador.

Preguntas básicas

1. ¿Qué es un ciclo?
2. ¿Cuáles son las instrucciones que pertenecen a la estructura ciclo?
3. ¿Cómo funciona la instrucción MIENTRAS?
4. ¿En qué consiste un contador?

Contenidos del módulo

13.1 Definición de estructura ciclo
13.2 Instrucción MIENTRAS
13.3 Entrada de datos usando un ciclo
13.4 Ejemplo de uso
13.5 Concepto de contador

«La repetición de actividades o tareas es una de las situaciones que se presentan frecuentemente en la vida diaria. Dicha repetición depende de alguna condición. Por ejemplo, en una tienda se atiende a los clientes siempre y cuando la tienda se encuentre en servicio. Por consiguiente podemos plantear un ciclo: mientras la tienda esté abierta, se atiende a los clientes».

Vea en el botón **MIENTRAS** del mapa conceptual el video "Módulo 13. Estructura ciclo".

13.1 Definición de estructura ciclo

La estructura ciclo está conformada por instrucciones que permiten al programador instruir la máquina para que ejecute varias veces un grupo de instrucciones según alguna condición. Las instrucciones que pertenecen a esta estructura son:

MIENTRAS, PARA y HAGA.

13.2 Instrucción MIENTRAS

La forma general de la instrucción MIENTRAS es:

```
MIENTRAS  condición
          Conjunto de instrucciones que se ejecutan mientras que la condición
          sea verdadera
Fin(MIENTRAS)
```

La condición puede ser una expresión relacional o una expresión lógica. En la condición, por lo general, se evalúa el valor de una variable. Esta variable, que se debe modificar en cada iteración, recibe el nombre de *variable controladora* del ciclo.

13.3 Entrada de datos usando un ciclo

Para elaborar un algoritmo con ciclos con el fin de determinar el aumento y el nuevo salario de todos los empleados de una empresa, según los criterios del algoritmo del módulo anterior, debemos primero definir cómo entrar los datos al algoritmo. Una forma es a través del teclado y otra forma es leyendo los datos de cada empleado desde un archivo previamente almacenado en el computador.

Vamos a elaborar nuestro algoritmo leyendo los datos desde un archivo. Como habíamos visto anteriormente, un archivo es una colección de registros, en la cual en cada registro se almacenan los datos correspondientes a un empleado. Cuando se procesa un archivo, el computador debe identificar el nombre del archivo y cuándo terminó de procesarlo. Para identificar esta segunda situación, cada archivo tiene al final un registro especial que indica que ya no hay más registros. Este registro especial contiene lo que se conoce como *marca de fin de archivo*. Llamemos a esta marca EOF, de las iniciales de las palabras en inglés **E**nd **o**f **F**ile (fin de archivo).

Por ejemplo, un archivo con cinco registros llamado '**nomina**', en el cual cada registro contiene el nombre y el salario actual de un empleado, tiene la siguiente conformación:

Archivo '**nómina**'

n.º de registro	Nombre	Salario actual
1	Sara	2000
2	Pedro	3000
3	María	1500
4	Juan	500
5	Elena	2800
6	EOF	

Cuando se está procesando un archivo y se ejecuta la instrucción de lectura LEA, la máquina lee los datos de un registro y se posiciona para leer los datos del siguiente registro. Por consiguiente, para procesar el anterior archivo debemos ejecutar la instrucción LEA cinco veces.

Además, para procesar un archivo necesitamos ponerlo a disposición del programa que lo va a procesar. Esto se logra con la instrucción ABRA(nombre del archivo). También incluiremos una pequeña modificación en la instrucción de lectura para señalar cuál es el archivo que se va a leer, y una instrucción que cierre el archivo cuando se termina de procesar.

Nuestra instrucción de lectura queda:

LEA(nombre del archivo: lista de variables, separadas por comas)

13.4 Ejemplo de uso

Considerando estas características, con respecto al manejo de archivos, nuestro algoritmo del módulo anterior queda:

```
1.   Algoritmo aumentoEnArchivo(1)
2.       Variables: sa, au, ns: numericaReal
3.                  nom: alfanumérica
4.       INICIO
5.           ABRA(nomina)
6.           MIENTRAS  no sea EOF(nomina)
7.               LEA(nomina: nom, sa)
8.               au = 0
9.               SI  sa < 1000
10.                  au = sa * 0.1
11.              Fin(SI)
12.              ns = sa + au
13.              ESCRIBA(nom, sa, au, ns)
14.          Fin(MIENTRAS)
15           CIERRE(nomina)
16.      FIN
17.  Fin(aumentoEnArchivo)
```

Al ejecutar la instrucción 5, el dispositivo de lectura se posiciona para leer el primer registro del archivo llamado '**nomina**'.

La instrucción 6 pregunta si el registro disponible para ser leído tiene la marca de fin de archivo; de ser así, la ejecución continuará en la instrucción 15.

Si el registro disponible para ser leído no es la marca de fin de archivo, la ejecución continúa con la instrucción 7.

La instrucción 7 lee nombre y salario y los almacena en las variables llamadas **nom** y **sa**: en la variable **nom** queda almacenado Sara y en la variable **sa** queda almacenado 2000.

Además de cargar las variables **nom** *y* **sa** *con los datos leídos, la instrucción de lectura deja posicionado el dispositivo de lectura para que lea el siguiente registro.*

En la instrucción 8 se le asigna 0 al aumento.

La instrucción 9 compara el salario leído con 1000. Si el salario leído es menor que 1000 ejecuta la instrucción 10, la cual calcula el aumento como el diez por

ciento del salario leído (**au** = **sa** * 0.1). Si el salario leído no es menor que 1000, el aumento se queda en cero.

En la instrucción 12 se calcula el nuevo salario, y en la 13 se escribe el nombre del empleado, el salario actual, el aumento y el nuevo salario.

La instrucción 14, que es el fin de la instrucción MIENTRAS, hace que el algoritmo retorne a la instrucción 6 en la cual va a controlar que el registro para leer no tenga la marca de fin de archivo.

Al ejecutar por segunda vez la instrucción LEA, las variables **nom** y **sa** quedan con los datos Pedro y 3000, respectivamente.

Nuevamente se le asigna 0 al valor del aumento, ya que si no hiciéramos esto en la variable **au** *quedará el aumento que se le hizo al anterior empleado.*

Al procesar el archivo **'nomina'** en su totalidad, los resultados obtenidos serán:

nom	sa	au	ns
Sara	2000	0	2000
Pedro	3000	0	3000
María	1500	0	1500
Juan	500	50	550
Elena	2800	0	2800

13.5 Concepto de contador

En muchas ocasiones es importante conocer, al final del proceso, cuántos registros se procesaron. Para ello utilizamos una variable adicional, la cual tendrá un valor inicial de 0, y cada vez que se lea un registro la incrementamos en 1. Una variable con esta característica se denomina CONTADOR.

Escribamos nuestro algoritmo *aumentoEnArchivo* contando los registros procesados e imprimiendo dicho total al final del proceso del archivo:

```
1.   Algoritmo aumentoEnArchivo(2)
2.       Variables: sa, au, ns: numericaReal
3.                  nom: alfanumérica
4.                  i: numericaEntera
5.       INICIO
6.           ABRA(nomina)
7.           i = 0
8.           MIENTRAS  no sea EOF(nomina)
9.               LEA(nomina: nom, sa)
10.              i = i + 1
11.              au = 0
12.              SI  sa < 1000
13.                  au = sa * 0.1
14.              Fin(SI)
15.              ns = sa + au
16.              ESCRIBA(sa, au, ns)
17.          Fin(MIENTRAS)
```

```
18.             CIERRE(nomina)
19.             ESCRIBA(“total empleados: “, i)
20.        FIN
21.   Fin(aumentoEnArchivo)
```

En este segundo algoritmo hemos introducido cuatro nuevas instrucciones: la instrucción 4, en la cual definimos la variable **i**, que hará las veces de contador; la instrucción 7, en la cual se inicializa el contador en 0; la instrucción 10, en la cual se incrementa el contador en 1, cada vez que se lee un registro; y la instrucción 19, en la cual se imprime el contenido del contador con su correspondiente mensaje.

Al ejecutar este segundo algoritmo con el mismo archivo **‘nomina’** el resultado que se obtiene es:

nom	sa	au	ns
Sara	2000	0	2000
Pedro	3000	0	3000
María	1500	0	1500
Juan	500	50	550
Elena	2800	0	2800

Total de empleados: 5

Resumen

En este módulo hemos presentado una introducción a la estructura ciclo con un ejemplo de aplicación usando la instrucción MIENTRAS.

Ejercicios propuestos

1. Elabore un algoritmo que procese un archivo llamado **‘edades’**, en el cual cada registro contiene el nombre de una persona y su edad en años. El algoritmo debe imprimir el nombre de la persona y su clasificación de acuerdo a los siguientes criterios: la persona es ‘niño’, si la edad está entre 0 y 10 años; ‘adolescente’, si la edad está entre 11 y 18 años; ‘adulto’, si la edad está entre 19 y 35 años; ‘maduro’, si la edad está entre 36 y 60 años; y ‘anciano’, si la edad es mayor de 60 años. Además, al final debe imprimir cuántas personas hay en cada clasificación y el total de personas.

2. Elabore un algoritmo que procese un archivo llamado **‘promedios’**, en el cual cada registro contiene el nombre de un estudiante y su promedio acumulado de la carrera. El algoritmo debe imprimir el nombre del estudiante y alguno de los siguientes mensajes: ‘pésimo’, ‘malo’, ‘regular’, ‘bueno’ o ‘excelente’. El estudiante se considera pésimo si el promedio acumulado es menor o igual que 1; malo, si el promedio es mayor que 1 y menor que 3; regular, si el promedio es mayor o igual que 3 y menor que 4; bueno, si el promedio es mayor o igual que 4 y menor que 4.5; y excelente, si el promedio es mayor o igual que 4.5. Al final el algoritmo debe imprimir el número de estudiantes en cada clasificación y el total de estudiantes.

3. Se tiene una competencia ciclística en la cual los competidores son clasificados por categorías dependiendo del sexo y la edad que tengan. Las categorías definidas son: categoría A, mujeres menores de 14 años; B, mujeres entre 14 y 20 años; C, mujeres entre 21 y 35 años; D, mujeres mayores de 35 años; E, hombres menores de 15 años; F, hombres entre 15 y 25 años; G, hombres entre 26 y 40 años; H, hombres mayores de 40 años. Para ello se ha grabado un archivo, llamado **'inscritos'**, en el cual cada registro contiene los siguientes datos: nombre, sexo y edad. El sexo está codificado así: 0 significa que es mujer, 1 significa que es hombre. Elabore un algoritmo que procese dicho archivo y que imprima cuántas personas hay inscritas en cada categoría, en cuál categoría se inscribieron más personas y el total de participantes inscritos.

4. Se tiene un archivo llamado **'definitivas'**, en el cual cada registro contiene el nombre de un estudiante y las tres notas correspondientes a las notas definitivas de las tres materias que cursó. Elabore un algoritmo que calcule e imprima el promedio de cada estudiante y que al final imprima el nombre del estudiante que obtuvo el mejor promedio y el nombre del estudiante que obtuvo el peor promedio.

Módulo 14

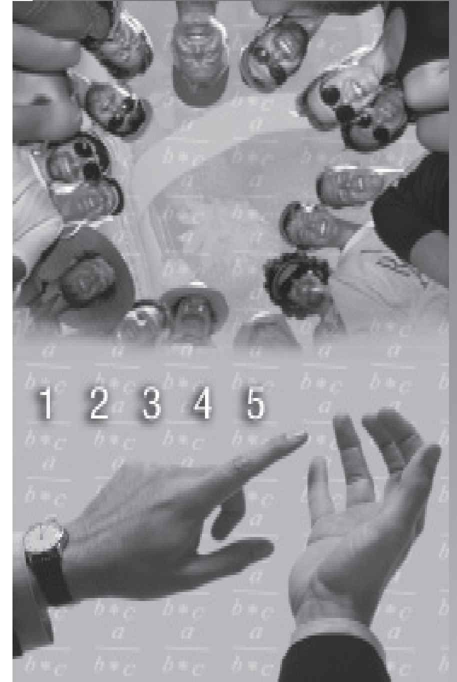
Contadores y acumuladores

Introducción
Cuando se trabaja un ciclo es necesario, en muchas ocasiones, contabilizar el total de ciertos datos que se procesan dentro del ciclo, para luego usarlos. Esta contabilización se efectúa usando variables que, en general, se denominan contadores y acumuladores.

Objetivos del módulo
1. Identificar los conceptos de contador y acumulador.
2. Aplicar y usar los conceptos de contador y acumulador.

Preguntas básicas
1. ¿Qué es un acumulador?
2. ¿Cómo se procesa un acumulador?
3. ¿En cuál parte del procesamiento se usa el resultado de un acumulador?

Contenidos del módulo
14.1 Concepto de acumulador
14.2 Ejemplo de algoritmo con acumuladores
14.3 Aplicación y uso de acumuladores



«Cuando se trata de actividades que se ejecutan cíclicamente, es necesario en muchas ocasiones manejar datos acerca de las actividades que se están desarrollando. Por ejemplo, al tendero le interesará conocer cuántos clientes ha atendido y cuál es el total de ventas. La variable que le dice cuántos clientes ha atendido será un contador, y la que le dice cuál es el total de ventas es un acumulador».

Vea en el botón **MIENTRAS** del mapa conceptual el video "Módulo 14. Intrucción MIENTRAS".

14.1 Concepto de acumulador

En el algoritmo del módulo anterior se presentó el concepto de contador, que es una variable que se utiliza, como su nombre lo dice, para contar las ocurrencias de algún evento. Es también necesario, en muchas ocasiones, conocer totales de ciertos datos, como total de salarios anteriores, total de aumentos y total de nuevos salarios.

Para contabilizar estos totales debemos manejar otras tres variables adicionales: una para llevar el total de salarios anteriores, otra para llevar el total de aumentos y otra para llevar el total de nuevos salarios. Llamemos a estas variables **tsa**, **tau** y **tns**, respectivamente. Inicialmente estas variables deben tener un valor de cero.

Cada vez que se lea un registro debemos incrementar el total de salarios anteriores con el salario leído:

```
tsa = tsa + sa
```

Cada vez que se calcule un nuevo aumento debemos incrementar el total de aumentos:

```
tau = tau + au
```

Cada vez que se calcule un nuevo salario debemos incrementar el total de nuevos salarios:

```
tns = tns + ns
```

Una variable con esta característica se denomina ACUMULADOR.

En nuestro ejemplo las variables **tsa**, **tau** y **tns** son entonces acumuladores.

14.2 Ejemplo de algoritmo con acumuladores

Nuestro algoritmo del módulo anterior, *aumentoEnArchivo*, quedará así:

```
1.    Algoritmo aumentoEnArchivo(3)
2.        Variables: sa, au, ns, tsa, tau, tns: numericaReal
3.                   nom: alfanumérica
4.                   i: numericaEntera
5.        INICIO
6.            tsa = 0
7.            tau = 0
8.            tns = 0
9.            i = 0
10.           ABRA(nomina)
11.           MIENTRAS  no sea EOF(nomina)
12.               LEA(nomina: nom, sa)
13.               tsa = tsa + sa
14.               i = i + 1
15.               au = 0
16.               SI  sa < 1000
17.                   au = sa * 0.1
18.                   tau = tau + au
19.               Fin(SI)
20.               ns = sa + au
21.               tns = tns + ns
```

```
22.                 ESCRIBA(sa, au, ns)
23.             Fin(MIENTRAS)
24.             CIERRE(nomina)
25.             ESCRIBA("total empleados: ", i)
26.             ESCRIBA("total salario anterior: ", tsa)
27.             ESCRIBA("total aumentos: ", tau)
28.             ESCRIBA("total nuevos salarios: ", tns)
29.         FIN
30.     Fin(aumentoEnArchivo)
```

En las instrucciones 6 a 8 se inicializan los acumuladores en cero.

En la instrucción 13 se acumula el salario actual. Esta instrucción se ejecuta en este sitio, puesto que todo empleado que se lee tiene un salario actual y el acumulador se debe actualizar tan pronto se tenga el dato al cual hay que llevarle un total.

La instrucción 18 acumula el total de aumentos. Esta instrucción se pone en este sitio, ya que ahí es donde se determina el aumento de cada empleado. Realmente esta instrucción se puede colocar fuera de la instrucción SI, pero es mejor colocarla adentro, ya que si lo hacemos después del fin del SI, en algunas ocasiones el resultado de efectuar esta instrucción es sumarle cero al acumulador **tau**, lo cual no afecta el contenido de **tau** pero implica ejecutar una operación.

En la instrucción 21 se acumula el total del nuevo salario, es decir, en la instrucción inmediatamente siguiente a la instrucción en la cual se calculó el nuevo salario.

Al ejecutar este algoritmo con los datos de nuestro archivo **'nomina'** los resultados que obtenemos son:

nom	sa	au	ns
Sara	2000	0	2000
Pedro	3000	0	3000
María	1500	0	1500
Juan	500	50	550
Elena	2800	0	2800

Total de empleados: 5
Total del salario anterior: 9800
Total de aumentos: 50
Total de nuevos salarios: 9850

14.3 Aplicación y uso de acumuladores

Otros datos de interés que se suelen calcular son los promedios de algunos datos. En nuestro ejercicio, por ejemplo, es interesante conocer los promedios de salario anterior, de aumento y de nuevo salario.

En general, para determinar un promedio basta con dividir un acumulador entre un contador. Así, en nuestro ejercicio el promedio de salario anterior se obtiene dividiendo el total de salario anterior entre el contador de empleados, el promedio de aumento se obtiene dividiendo el total de aumento entre el contador de empleados y el promedio del nuevo

salario se obtiene dividiendo el total del nuevo salario entre el contador de empleados. Llamemos a estas variables para promedios **psa**, **pau** y **pns.**

Nuestro algoritmo *aumentoEnArchivo* queda:

```
1.    Algoritmo aumentoEnArchivo(4)
2.        Variables: sa, au, ns, tsa, tau, tns, psa, pau, pns: numericaReal
3.                   nom: alfanumérica
4.                   i: numericaEntera
5.        INICIO
6.            tsa = 0
7.            tau = 0
8.            tns = 0
9.            i = 0
10.           ABRA(nomina)
11.           MIENTRAS  no sea EOF(nomina)
12.               LEA(nomina: nom, sa)
13.               tsa = tsa + sa
14.               i = i + 1
15.               au = 0
16.               SI  sa < 1000
17.                   au = sa * 0.1
18.                   tau = tau + au
19.               Fin(SI)
20.               ns = sa + au
21.               tns = tns + ns
22.               ESCRIBA(sa, au, ns)
23.           Fin(MIENTRAS)
24.           CIERRE(nomina)
25.           psa = tsa / i
26.           pau = tau / i
27.           pns = tns / i
28.           ESCRIBA("total empleados: ", i)
29.           ESCRIBA("total salario anterior: ", tsa, " promedio: ", psa)
30.           ESCRIBA("total aumentos: ", tau, " promedio: ", pau)
31.           ESCRIBA("total nuevos salarios: ", tns, " promedio: ", pns)
32.       FIN
33.   Fin(aumentoEnArchivo)
```

Los promedios se calculan en las instrucciones 25, 26 y 27.

Nota: *los promedios se calculan después de que se ha terminado de ejecutar el ciclo MIENTRAS***.**

Es un error común en los principiantes calcular promedios dentro del ciclo. Los promedios se calculan cuando se tiene el total de los contadores y el total de los acumuladores, y estos datos sólo se tienen cuando se ha terminado el ciclo.

Al ejecutar este último algoritmo con nuestro archivo **'nomina'** los resultados que obtenemos son los siguientes:

nom	sa	au	ns
Sara	2000	0	2000
Pedro	3000	0	3000
María	1500	0	1500
Juan	500	50	550
Elena	2800	0	2800

Total de empleados: 5
Total del salario anterior: 9800   promedio: 1960
Total de aumentos: 50   promedio: 10
Total de nuevos salarios: 9850   promedio: 1970

Resumen

En este módulo presentamos los conceptos de contador y acumulador, que son de gran utilidad en la elaboración de algoritmos.

Ejercicios propuestos

1. Elabore un algoritmo que procese un archivo llamado **'edades'**, en el cual cada registro contiene el nombre de una persona y su edad en años. El algoritmo debe imprimir el nombre de la persona y su clasificación de acuerdo a los siguientes criterios: la persona es 'niño', si la edad está entre 0 y 10 años; 'adolescente', si la edad está entre 11 y 18 años; 'adulto', si la edad está entre 19 y 35 años; 'maduro', si la edad está entre 36 y 60; y 'anciano', si la edad es mayor de 60 años. Además, al final debe imprimir cuántas personas hay en cada clasificación, el total de personas en el archivo, el promedio de edad en cada categoría y el promedio de edad de todas las personas.

2. Elabore un algoritmo que procese un archivo llamado **'promedios'**, en el cual cada registro contiene el nombre de un estudiante y su promedio acumulado de la carrera. El algoritmo debe imprimir el nombre del estudiante y alguno de los siguientes mensajes: 'pésimo', 'malo', 'regular', 'bueno' o 'excelente'. El estudiante se considera pésimo si el promedio acumulado es menor o igual que 1; malo, si el promedio es mayor que 1 y menor que 3; regular, si el promedio es mayor o igual que 3 y menor que 4; bueno, si el promedio es mayor o igual que 4 y menor que 4.5; y excelente, si el promedio es mayor o igual que 4.5. Al final el algoritmo debe imprimir el número de estudiantes en cada clasificación, el total de estudiantes, el promedio de nota de los estudiantes que quedaron en cada mensaje y el promedio total de notas.

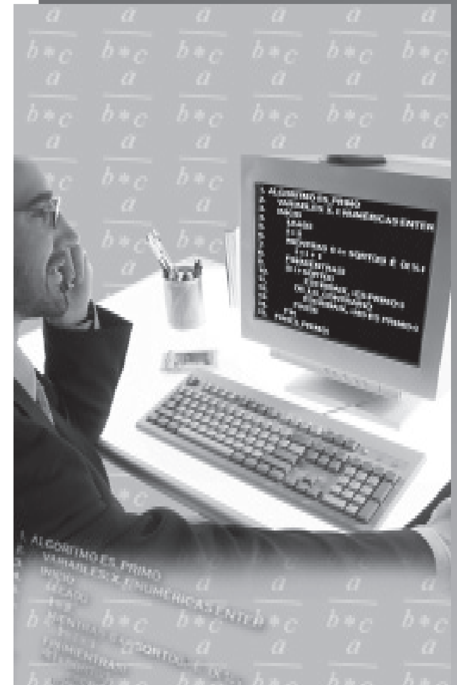
3. Se tiene una competencia ciclística en la cual los competidores son clasicados por categorías dependiendo del sexo y la edad que tengan. Las categorías definidas son: categoría A, mujeres menores de 14 años; B, mujeres entre 14 y 20 años; C, mujeres entre 21 y 35 años; D, mujeres mayores de 35 años; E, hombres menores de 15 años; F, hombres entre 15 y 25 años; G, hombres entre 26 y 40 años; H, hombres mayores de 40 años. Para ello se ha grabado un archivo, llamado **'inscritos'**, en el cual cada registro contiene los siguientes datos: nombre, sexo y edad. El sexo está codificado así: 0 significa que es mujer, 1 significa que es hombre. Elabore un algoritmo que procese dicho archivo y que imprima cuántas personas hay inscritas en cada categoría,

en cuál categoría se inscribieron más personas, el total de participantes inscritos y el promedio de edad de cada categoría.

4. Se tiene un archivo llamado **‘definitivas’**, en el cual cada registro contiene el nombre de un estudiante y las tres notas correspondientes a las notas definitivas de las tres materias que cursó. Elabore un algoritmo que calcule e imprima el promedio de cada estudiante y que al final imprima el nombre del estudiante que obtuvo el mejor promedio, el nombre del estudiante que obtuvo el peor promedio y el promedio de cada una de las materias que cursaron los estudiantes.

Módulo 15

Aplicaciones de ciclos en matemáticas



Introducción

Hasta aquí sólo hemos tratado ejercicios que se conocen como algoritmos de corte comercial y en los cuales se procesa un archivo. Los computadores son útiles para dar solución no sólo a problemas de carácter comercial, sino también a problemas de carácter científico y matemático. En este módulo trataremos un ejercicio que cae en este último rango.

Objetivos del módulo

1. Aplicar la instrucción MIENTRAS en ejercicios de tipo matemático.

Preguntas básicas

1. ¿En qué se diferencia una aplicación comercial de una aplicación matemática?

Contenidos del módulo

15.1 Enunciado de ejercicio de aplicación
15.2 Elaboración del algoritmo
15.3 Prueba de escritorio

«El campo de las matemáticas es un área en la cual la ayuda del computador es esencial. Efectuar cálculos matemáticos es una labor en la que la repetición de operaciones con datos diferentes es muy frecuente. Más aún, muchas de las labores comerciales requieren conceptos matemáticos y la manipulación de dichos conceptos».

Vea en el botón **MIENTRAS** del mapa conceptual el video "Módulo 15. Ejercicio con la instrucción MIENTRAS".

15.1 Enunciado de ejercicio de aplicación

Consideremos un algoritmo que lea un dato numérico entero y que determine si es número primo o no.

Recordemos que número primo es aquel que sólo es divisible por sí mismo y por la unidad. Teniendo en cuenta esta característica, una de las formas de saber si un número **x** es primo o no es dividiéndolo por todos los enteros desde 2 hasta la raíz cuadrada de **x**. Si alguna división es exacta, es decir, el residuo de la división es cero, **x** no es primo; pero si en todas las divisiones el residuo es diferente de cero, entonces **x** es primo.

Recordemos que el operador módulo (%) halla el residuo de una división entera.

15.2 Elaboración del algoritmo

El análisis para nuestro algoritmo es:

1. Datos de entrada:
   - Número entero: **x**

2. Datos de salida:
   - El mensaje "es primo" o "no es primo"

3. Cálculos:
   - Hallar el residuo de dividir **x** por todos los enteros desde 2 hasta $\sqrt[2]{x}$, o hasta que algún residuo dé igual a cero. Estos cálculos implican generar un divisor, que lo representaremos con la variable **i**.

```
1.   Algoritmo esPrimo
2.        Variables: x, i: numericaEntera
3.        INICIO
4.             LEA(x)
5.             i = 2
6.             MIENTRAS  (i <= sqrt(x)) && (x % i  ≠  0)
7.                  i = i + 1
8.             Fin(MIENTRAS)
9.             SI i > sqrt(x)
10.                 ESCRIBA(x, " es primo")
11.            DE_LO_CONTRARIO
12.                 ESCRIBA(x, " no es primo")
13.            Fin(SI)
14.       FIN
15.  Fin(esPrimo)
```

En la instrucción 4 se lee el número que hay que averiguar si es primo o no.

En la instrucción 5 se inicializa una variable, llamada **i**, que nos va a servir de divisor. Esta variable tomará los valores desde 2 hasta la raíz cuadrada de **x**.

En la instrucción 6 se plantea el ciclo. La instrucción de ciclo controla dos situaciones: una, que **i** sea menor o igual que la raíz cuadrada de **x**, y otra, que el residuo de dividir **x** entre **i** sea diferente de cero. Para controlar que **i** sea menor o igual que la raíz cuadrada de **x** utilizamos un programa proporcionado por el computador, llamado sqrt(), el cual halla la raíz cuadrada de un número. (Más adelante veremos

los programas que suministra el computador.) Para controlar que el residuo de la división de **x** por un número **i** sea diferente de cero utilizamos el operador módulo (%). Cuando se ejecuta la instrucción MIENTRAS, se evalúan las dos situaciones. Si ambas condiciones son verdaderas, se ejecuta la instrucción del ciclo. Con una de las dos condiciones que sea falsa no entra a ejecutar las instrucciones del ciclo y continúa con la instrucción siguiente al fin del MIENTRAS.

Cuando se sale del ciclo es porque alguna de las dos condiciones es falsa: si es falso que **i** sea menor o igual que la raíz cuadrada de **x** significa que dividió **x** por todos los enteros entre 2 y la raíz cuadrada de **x** y que todas las divisiones arrojaron un residuo diferente de cero; por lo tanto, el número **x** es primo. Si se sale del ciclo porque el residuo de dividir **x** entre **i** es cero (x % i == 0), significa que **x** no es un número primo. Estas situaciones se controlan en las instrucciones 9 a 13.

15.3 Prueba de escritorio

Veamos cómo se comporta nuestro algoritmo cuando el número al cual deseamos averiguar si es primo o no es el número 11.

Al ejecutar la instrucción 4 el contenido de **x** será 11.

En la instrucción 5 se inicializa la variable **i** en 2.

La ejecución de la instrucción 6 implica calcular la raíz cuadrada de **x** y el residuo de dividir **x** entre **i**: la raíz cuadrada de 11 es 3.3 y el residuo de dividir 11 entre 2 es 1. Ambas condiciones son verdaderas, por consiguiente entra al ciclo e incrementa el valor de **i** en 1: **i** queda valiendo 3.

Se llega al fin del MIENTRAS, entonces regresa a evaluar de nuevo las dos condiciones: **i** sigue siendo menor que la raíz cuadrada de 11 y el residuo de dividir 11 entre 3 es 2: ambas condiciones son verdaderas, por lo tanto vuelve y entra al ciclo e incrementa **i** nuevamente en 1: **i** queda valiendo 4.

Llega otra vez al fin del MIENTRAS y retorna a evaluar nuevamente las condiciones: el contenido de la variable **i** ya no es menor que la raíz cuadrada de 11, por consiguiente ya no entrará al ciclo sino que continuará con la instrucción siguiente al fin del MIENTRAS: la instrucción 9.

En la instrucción 9 se compara el contenido de la variable **i** con la raíz cuadrada de **x**. En nuestro caso **i** es mayor que la raíz cuadrada de **x**, por lo tanto imprime el contenido de **x** (11) con el mensaje “es primo”.

Hagámoslo ahora con el número 16.

Al ejecutar la instrucción 4 el contenido de **x** será 16.

En la instrucción 5 se inicializa la variable **i** en 2.

La ejecución de la instrucción 6 implica calcular la raíz cuadrada de **x** y el residuo de dividir **x** entre **i**: la raíz cuadrada de 16 es 4 y el residuo de dividir 16 entre 2 es 0. La primera condición es verdadera, pero la segunda es falsa, por consiguiente no entra al ciclo MIENTRAS y continúa la ejecución con la instrucción 9.

En la instrucción 9 se compara el contenido de la variable **i** con la raíz cuadrada de **x**. En nuestro segundo ejemplo **i** es menor que la raíz cuadrada de **x**, por lo tanto imprime el contenido de **x** (16) con el mensaje “no es primo”.

Resumen

En este módulo hemos presentado una aplicación de la instrucción de ciclo MIENTRAS en un ejercicio de carácter matemático, en el cual usamos también las instrucciones que se habían desarrollado con anterioridad.

Ejercicios propuestos

1. Elabore un algoritmo que lea un entero **n** y que imprima todos los múltiplos de 3 desde 1 hasta **n**.

2. Elabore un algoritmo que lea un entero **n** menor que 32768 y que imprima la suma de sus dígitos.

3. Elabore un algoritmo que lea un entero **n** menor que 32768 y que invierta sus dígitos. Por ejemplo, si el número leído es 31628, el algoritmo debe imprimir 31628 y 82613.

4. Elabore un algoritmo que lea dos enteros positivos y que imprima todos los números impares entre el menor y el mayor de los números leídos.

5. Elabore un algoritmo que lea dos números enteros y que determine e imprima el máximo común divisor y el mínimo común múltiplo de ellos.

6. Elabore un algoritmo que lea un número entero **n** y que imprima todos los divisores de él.

7. Elabore un algoritmo que lea un número positivo y que calcule e imprima la raíz cuadrada del número, sin utilizar los programas que suministra el computador.

8. Elabore un algoritmo que lea un entero **n** y que genere los **n** primeros términos de la serie de Fibonacci. La serie de Fibonacci se construye partiendo de que los dos primeros términos son el 0 y el 1, y cada siguiente término se calcula como la suma de los dos anteriores. Ejemplo: 0, 1, 1, 2, 3, 5, 8, 13, …

9. Elabore un algoritmo que lea un entero **n** y que calcule e imprima la suma de los **n** primeros términos de la serie de Fibonacci.

10. Elabore un algoritmo que lea un entero positivo **n** y que calcule e imprima la sumatoria de todos los enteros desde 1 hasta **n**.

11. Elabore un algoritmo que lea un entero **n** y que determine e imprima si es un número perfecto. Número perfecto es aquel cuya suma de sus divisores desde 1 hasta **n – 1** es **n**.

12. Elabore un algoritmo que lea un entero positivo **n** y que calcule e imprima el factorial de **n**. El factorial de un entero es el producto de todos los dígitos desde 1 hasta **n**. Por ejemplo, si **n** es 5, el factorial de 5 es 1 * 2 * 3 * 4 * 5 = 120 y se escribe 5! = 120.

13. Elabore un algoritmo que lea un entero **n** y que calcule e imprima el valor de los primeros **n** términos de la siguiente serie:

$$X - \frac{X^3}{3} + \frac{X^5}{5} - \frac{X^7}{7} + \frac{X^9}{9} - \ldots$$

14. Elabore un algoritmo que lea un entero **n** y que calcule e imprima el valor de los primeros **n** términos de la siguiente serie:

$$X - \frac{X}{1!} + \frac{X^2}{2!} - \frac{X^3}{3!} + \frac{X^4}{4!} - \ldots$$

15. Elabore un algoritmo que lea dos enteros **m** y **n**, y que calcule e imprima el cociente de dividir **m** entre **n**, utilizando únicamente operaciones de suma y resta.

16. Elabore un algoritmo que lea un entero **n** y que determine e imprima el número de dígitos del número leído.

17. Elabore un algoritmo que lea dos números enteros y que imprima el que tenga más dígitos. Si el número de dígitos es el mismo debe imprimir el mayor de los dos números.

18. Elabore un algoritmo que lea dos números enteros y que imprima todos los enteros terminados en 7 que haya entre los dos números leídos.

19. Elabore un algoritmo que determine e imprima el número primo más cercano a 32768 y menor que 32768.

Módulo 16

Ciclos anidados con instrucción MIENTRAS

Introducción

Hasta el momento hemos venido tratando algoritmos en los cuales utilizamos la instrucción de ciclo MIENTRAS de una forma sencilla, es decir, el conjunto de instrucciones que se ejecutan repetitivamente está conformado por sólo instrucciones de asignación, instrucciones de lectura, instrucciones de escritura e instrucciones correspondientes a la estructura decisión. Hay situaciones en las cuales es necesario ejecutar ciclos dentro de un ciclo. Esta situación se conoce como ciclos anidados. Trataremos esta situación con ejercicios de carácter matemático.

Objetivos del módulo

1. Usar la instrucción MIENTRAS en ejercicios de carácter matemático.

Preguntas básicas

1. ¿Qué es un ciclo anidado?
2. ¿Cómo funcionan los ciclos anidados?
3. ¿Cuándo se requieren ciclos anidados?

Contenidos del módulo

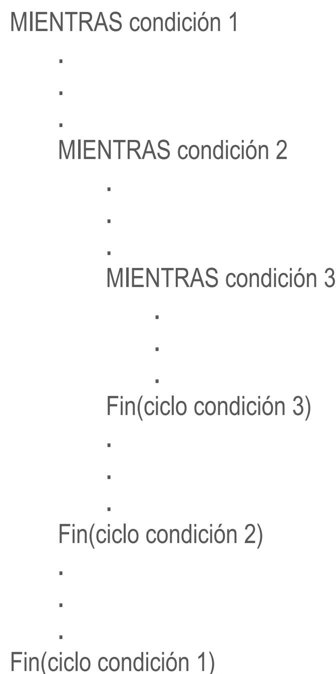
16.1 Definición
16.2 Algoritmo de una tabla de multiplicar
16.3 Algoritmo de varias tablas de multiplicar

«En algunas situaciones se ejecutan actividades cíclicas dentro de otros ciclos. En el caso de una tienda, atender a un cliente puede ser una actividad cíclica dentro de un ciclo planteado inicialmente: mientras la tienda esté abierta, se atiende a los clientes. Entonces, dentro de ese ciclo, se puede plantear este otro: mientras esté atendiendo a un cliente, no atiendo a otro cliente».

Vea en el botón **MIENTRAS** del mapa conceptual el video "Módulo 16. Ciclos anidados".

16.1 Definición

Se define como ciclos anidados la estructura en la cual un ciclo se halla completamente dentro de otro ciclo. La forma general de esta estructura es:



La característica principal que debemos tener en cuenta en esta estructura es que para que se pueda continuar ejecutando un ciclo externo se deben haber terminado de ejecutar completamente sus ciclos internos. Cada vez que continúa la ejecución de un ciclo externo, volverá a ejecutar totalmente sus ciclos internos.

16.2 Algoritmo de una tabla de multiplicar

Elaborar un algoritmo que lea un entero **x** y que calcule e imprima la tabla de multiplicar de **x**, desde 1 hasta 10.

El análisis de nuestro algoritmo es:

1. Datos de entrada:
    - Número entero: **x**

2. Datos de salida:
    - La tabla de multiplicar de **x** desde 1 hasta 10.

3. Cálculos:
    - Generar los enteros desde 1 hasta 10, multiplicar cada entero generado por **x** e imprimir **x**, el entero generado y el resultado de la multiplicación.

Nuestro algoritmo es:

```
1.  Algoritmo tablaDeMultiplicarDeX
2.          Variables: x, i, r: numericaEntera
```

```
3.      INICIO
4.          LEA(x)
5.          i = 1
6.          MIENTRAS  i <= 10
7.              r = x * i
8.              ESCRIBA(x, "*", i, "=", r)
9.              i = i + 1
10.         Fin(MIENTRAS)
11.     FIN
12. Fin(tablaDeMultiplicarDeX)
```

En la instrucción 2 definimos las variables **x**, **i** y **r**. Con la variable **i** vamos a generar los enteros con los cuales multiplicaremos a **x**; en **r** guardaremos el resultado de cada multiplicación.

En la instrucción 4 leemos el valor de **x**.

En la instrucción 5 inicializamos la variable **i** en 1.

En la instrucción 6 controlamos que el valor de **i** no sea mayor que 10. Si el contenido de **i** es menor o igual que 10 se ejecutan las instrucciones 7 a 10.

En la instrucción 7 se multiplica el contenido de **x** por el contenido de **i**, y se almacena el resultado en la variable **r**.

En la instrucción 8 se escribe el contenido de **x**, el contenido de **i** y el contenido de **r** con los símbolos que indican que se efectuó un producto.

En la instrucción 9 se incrementa el valor de **i** en 1.

La instrucción 10 es el fin del MIENTRAS, lo cual hace que el programa se devuelva hacia la instrucción 6 para controlar el valor de **i**: si **i** es mayor que 10 no ejecutará las instrucciones del ciclo, sino que continuará con la ejecución de la instrucción 11.

La instrucción 11 es sólo la instrucción que indica que se terminó la ejecución del algoritmo.

Veamos cómo trabaja nuestro algoritmo cuando se lee un 5: **x** queda valiendo 5.

En la instrucción 5 inicializamos **i** en 1.

Al ejecutar la instrucción 6, en nuestro ejemplo **i** está valiendo 1, por lo tanto se ejecutan las instrucciones 7 a 10: en la instrucción 7 multiplica el contenido de **x** por el contenido de **i**, y el resultado lo almacena en la variable **r**: **r** queda valiendo 5; en la instrucción 8 escribe:

5 * 1 = 5

En la instrucción 9 se incrementa el valor de **i**, por lo tanto **i** queda valiendo 2.

La instrucción 10 lo regresa a la instrucción 6 en la cual vuelve y compara el contenido de **i** con 10: **i** vale 2, lo cual indica que la condición es verdadera y por consiguiente volverá a ejecutar las instrucciones del ciclo.

Cuando **i** tome el valor de 11 y regrese a ejecutar la instrucción 6 la condición será falsa, por lo tanto no ejecutará las instrucciones 7 a 10 y termina la ejecución del algoritmo.

Los resultados que imprime al ejecutar completamente el algoritmo, con **x** valiendo 5, es:

5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50

En el anterior ejemplo elaboramos un algoritmo que lee un entero **x** y calcula e imprime la tabla de multiplicar de **x** desde 1 hasta 10. Consideremos un ejemplo más completo.

16.3 Algoritmo de varias tablas de multiplicar

Elaborar un algoritmo que lea un entero **n** y que calcule e imprima todas las tablas de multiplicar desde 1 hasta **n**, cada tabla desde 1 hasta 10.

En este caso, si el número leído es 5, debemos generar e imprimir: la tabla de multiplicar del 1, desde 1 hasta 10; la tabla de multiplicar del 2, desde 1 hasta 10; la tabla de multiplicar del 3, desde 1 hasta 10; la tabla de multiplicar del 4, desde 1 hasta 10; y la tabla de multiplicar del 5, desde 1 hasta 10.

En nuestro algoritmo anterior leíamos un número **x** y generábamos e imprimíamos la tabla de multiplicar de **x**, desde 1 hasta 10. Ahora lo que necesitamos es generar las **x** desde 1 hasta 5, y para cada valor de **x** escribir las instrucciones correspondientes al algoritmo anterior. Nuestro algoritmo será:

Análisis:

1. Datos de entrada:
   - Número entero: **n**

2. Datos de salida:
   - Tablas de multiplicar desde 1 hasta **n**, cada tabla desde 1 hasta 10.

3. Cálculos:
   - Generar todos los enteros desde 1 hasta **n**: utilizaremos una variable **x**, la cual variará desde 1 hasta **n**, y por cada valor de **x** generamos e imprimimos la tabla de multiplicar desde 1 hasta 10.

Nuestro algoritmo es:

```
1.    Algoritmo tablasDeMultiplicar
2.        Variables: n, x, i, r: numericaEntera
3.        INICIO
4.            LEA(n)
5.            x = 1
6.            MIENTRAS  x <=  n
7.                i = 1
8.                MIENTRAS  i <= 10
9.                    r = x * i
10.                   ESCRIBA(r, "*", x, "=", i)
11.                   i = i + 1
12.               Fin(MIENTRAS)
13.               x = x + 1
14.           Fin(MIENTRAS)
15.       FIN
16.   Fin(tablasDeMultiplicar)
```

En la instrucción 4 se lee el valor de **n**.

En la instrucción 5 se inicializa el valor de **x** en 1.

En la instrucción 6 se controla que el valor de **x** sea menor o igual que **n**. Si el valor de **x** es menor o igual que **n**, se ejecutan las instrucciones 7 a 14, de lo contrario continúa con la ejecución de la 15.

Las instrucciones 7 a 12 son las mismas que las instrucciones 5 a 10 del algoritmo *tablaDeMultiplicarDeX*. Es decir, las instrucciones para generar la tabla de multiplicar de un número **x**. En otras palabras, hemos incrustado las instrucciones para generar la tabla de multiplicar de un número **x** en un ciclo, en el cual vamos a generar **x** desde 1 hasta **n**: por cada valor de **x** se genera e imprime la tabla de multiplicar desde 1 hasta 10.

Si leemos para **n** un valor de 5, el resultado obtenido al ejecutar el algoritmo *tablasDeMultiplicar* es:

```
1 * 1 = 1     2 * 1 = 2     3 * 1 = 3     4 * 1 = 4     5 * 1 = 5
1 * 2 = 2     2 * 2 = 4     3 * 2 = 6     4 * 2 = 8     5 * 2 = 10
1 * 3 = 3     2 * 3 = 6     3 * 3 = 9     4 * 3 = 12    5 * 3 = 15
1 * 4 = 4     2 * 4 = 8     3 * 4 = 12    4 * 4 = 16    5 * 4 = 20
1 * 5 = 5     2 * 5 = 10    3 * 5 = 15    4 * 5 = 20    5 * 5 = 25
1 * 6 = 6     2 * 6 = 12    3 * 6 = 18    4 * 6 = 24    5 * 6 = 30
1 * 7 = 7     2 * 7 = 14    3 * 7 = 21    4 * 7 = 28    5 * 7 = 35
1 * 8 = 8     2 * 8 = 16    3 * 8 = 24    4 * 8 = 32    5 * 8 = 40
1 * 9 = 9     2 * 9 = 18    3 * 9 = 27    4 * 9 = 36    5 * 9 = 45
1 * 10 = 10   2 * 10 = 20   3 * 10 = 30   4 * 10 = 40   5 * 10 = 50
```

Resumen

En este módulo hemos presentado un ejercicio más avanzado con la utilización de instrucciones MIENTRAS anidadas, es decir, ciclos dentro de ciclos.

Ejercicios propuestos

1. Elabore un algoritmo que lea un entero **n** y que genere e imprima todos los números primos desde 1 hasta **n**.

2. Elabore un algoritmo que lea dos enteros positivos **m** y **n**, y que calcule e imprima el resultado de multiplicar **m** por **n** utilizando únicamente la operación de suma.

3. Elabore un algoritmo que lea dos enteros positivos **m** y **n**, y que calcule e imprima el resultado de elevar **m** a la potencia **n** utilizando únicamente la operación de suma.

4. Elabore un algoritmo que lea un entero positivo **n** y que calcule e imprima 2^n utilizando sólo la operación de suma.

5. Elabore un algoritmo que imprima los enteros desde 1 hasta **n** de la siguiente manera:

   1 22 333 4444 55555 …

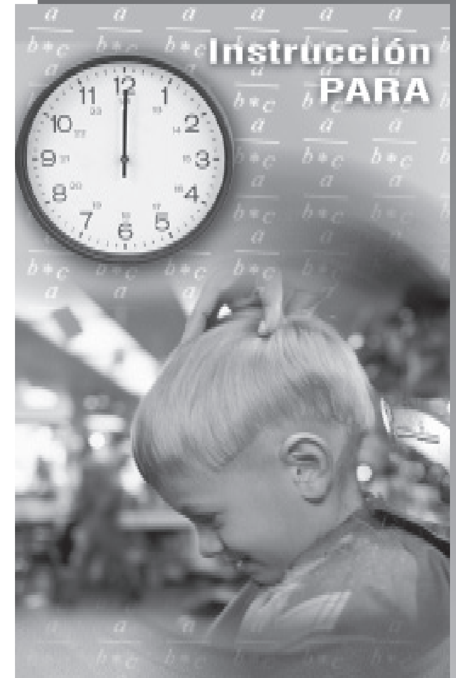
6. Una empresa utiliza la siguiente fórmula para calcular el sueldo de sus empleados:

   Sueldo = (100 + edad + (1 + 2 + 3 +...+ años en la compañía))/años en la compañía.

   Elabore un programa que permita imprimir el sueldo y el nombre de cada uno de los 40 empleados de la compañía, así como el total acumulado de sueldos y el nombre del empleado que gana más y del que gana menos.

Módulo 17

Instrucción PARA



Introducción

Como habíamos mencionado en el módulo 13 las instrucciones de ciclo son MIENTRAS, PARA y HAGA. En los módulos anteriores hemos tratado la instrucción MIENTRAS. En este trataremos la instrucción PARA, la cual tiene la característica que dentro de la misma instrucción se definen el valor inicial de la variable controladora del ciclo, el valor final y la variación de la misma.

Objetivos del módulo

1. Reconocer y aplicar la instrucción de ciclo PARA.
2. Diferenciar el uso de la instrucción PARA con la instrucción MIENTRAS.

Preguntas básicas

1. ¿Cuál es la forma general de la instrucción PARA?
2. ¿Cuál es la principal diferencia entre la instrucción PARA y la instrucción MIENTRAS?
3. ¿Cuándo es aconsejable utilizar la instrucción PARA?

Contenidos del módulo

17.1 Forma general
17.2 Primer ejemplo
17.3 Segundo ejemplo (variación positiva diferente de 1)
17.4 Tercer ejemplo (variación negativa)
17.5 Cuarto ejemplo (variación con variable)
17.6 Instrucción PARA *versus* instrucción MIENTRAS

«Hay situaciones en las cuales el ciclo tiene una duración fija. Consideremos por ejemplo una barbería en la cual el barbero sólo tiene un horario fijo de atención. Aquí podremos plantear un ciclo, con una duración predeterminada: se atiende a los clientes desde las 10 de la mañana hasta las 4 de la tarde».

Vea en el botón **PARA** del mapa conceptual el video "Módulo 17. Instrucción PARA".

17.1 Forma general

La forma general de la instrucción PARA es:

```
PARA  id DESDE v/r inicial HASTA v/r final CON_VARIACION [variable o constante]
      Conjunto de instrucciones que se ejecuta mientras id esté entre el valor inicial y
      el valor final.
Fin(PARA)
```

La instrucción PARA es una instrucción de ciclo que se utiliza cuando se conocen los valores inicial y final de la variable controladora del ciclo (**id**), y la variación de ella es constante.

17.2 Primer ejemplo

Elabore un algoritmo que imprima los enteros del 1 al 100.

```
1.   Algoritmo imprimeTodos
2.        Variables: i: numericaEntera
3.        INICIO
4.             PARA  i  DESDE  1  HASTA  100  CON_VARIACION  +1
5.                  ESCRIBA(i)
6.             Fin(PARA)
7.        FIN
8.   Fin(imprimeTodos)
```

En la instrucción 2 definimos la variable **i**, con la cual generaremos todos los enteros desde 1 hasta 100.

En la instrucción 4 utilizamos la instrucción PARA. Cuando se ejecuta la parte DESDE significa que a la variable **i** se le asigna un valor de 1; la parte HASTA indica que el mayor valor que podrá tomar **i** es 100, y la parte CON_VARIACION instruye al computador para que cuando se llegue al fin del PARA la variable **i** se incremente en 1.

Cada vez que llega al fin del PARA, incrementa la variable **i** en 1 y regresa a la instrucción 4 para controlar que el valor de la **i** no sea mayor que 100. Cuando el contenido de la variable **i** sea mayor que 100 termina de ejecutar las instrucciones del ciclo y continúa con la instrucción siguiente al fin del PARA.

El resultado de ejecutar este algoritmo es que imprime todos los enteros desde 1 hasta 100.

17.3 Segundo ejemplo (variación positiva diferente de 1)

Un segundo ejemplo es elaborar un algoritmo que imprima todos los enteros impares desde 1 hasta 100.

```
1.   Algoritmo imprimeImpares
2.        Variables: i: numericaEntera
3.        INICIO
4.             PARA  i  DESDE  1  HASTA  100  CON_VARIACION  +2
5.                  ESCRIBA(i)
6.             Fin(PARA)
7.        FIN
8.   Fin(imprimeImpares)
```

La única diferencia de este algoritmo con el primero es que en la instrucción 4 la variación es +2, lo cual instruye a la máquina para que cuando se llegue al fin del PARA incremente el contenido de la variable **i** en 2, logrando de esta manera que la **i** sólo tome los valores impares: 1, 3, 5, etc.

Una característica adicional de la instrucción PARA es que podemos modificar la variable controladora del ciclo en forma negativa.

17.4 Tercer ejemplo (variación negativa)

Elabore un algoritmo que imprima todos los enteros desde 100 hasta 1.

```
1.  Algoritmo imprimeDescendentemente
2.      Variables: i: numericaEntera
3.      INICIO
4.          PARA  i  DESDE  100  HASTA  1  CON_VARIACION  – 1
5.              ESCRIBA(i)
6.          Fin(PARA)
7       FIN
8.  Fin(imprimeDescendentemente)
```

En este tercer ejemplo la variable **i** toma un valor inicial de 100 y el valor final es 1. La parte CON_VARIACION es –1, la cual instruye a la máquina para que cuando se llegue al fin del PARA disminuya el contenido de la variable **i** en 1.

17.5 Cuarto ejemplo (variación con variable)

La forma como se modifica la variable controladora del ciclo puede ser una variable: elaborar un algoritmo que imprima todos los enteros comprendidos entre dos enteros leídos, con una variación también leída.

```
1.  Algoritmo imprimeNumeros
2.      Variables: in, fi, i, var: numericaEntera
3.      INICIO
4.          LEA(in, fi, var)
5.          PARA  i  DESDE  in  HASTA  fi  CON_VARIACION  var
6.              ESCRIBA(i)
7.          Fin(PARA)
8.      FIN
9.  Fin(imprimeNumeros)
```

En este ejemplo leemos tres datos, los cuales estamos llamando **in**, **fi** y **var**. En la variable **in** manejamos un entero que utilizaremos como valor inicial, en la variable **fi** manejamos otro entero que utilizaremos como valor final, y la variable **var** indicará cómo se modificará la variable controladora del ciclo.

Consideremos que los valores leídos son 15, 78 y 6.

> La variable **in** queda valiendo 15, la variable **fi** queda valiendo 78 y la variable **var** queda valiendo 6.

Al ejecutar la instrucción 5 la variable **i** toma un valor de 15, el cual compara con el valor final que es 78; el resultado es verdadero y por lo tanto ejecuta la instrucción 6, en la cual escribe el contenido de la variable **i**, que es 15.

Al ejecutar la instrucción 7, la cual es el fin del PARA, incrementa la variable **i** en 6 y regresa a la instrucción 5 a comparar el valor de **i** con el valor final que es 78. Como el contenido de la variable **i** es 21, que es menor que 78, ejecuta de nuevo la instrucción 6, imprimiendo el número 21.

De nuevo llega a la instrucción 7, incrementa el valor de la **i** en 6 y regresa a ejecutar nuevamente la instrucción 5.

Este proceso lo repite hasta que el valor de la **i** sea mayor que 78.

El resultado de ejecutar este algoritmo es que imprime los números:

15, 21, 27, 33, 39, 45, 51, 57, 63, 69, 75

Nota: *Cuando la variación es positiva el valor final debe ser mayor o igual que el valor inicial. Cuando la variación es negativa el valor final debe ser menor o igual que el valor inicial.*

Si no se cumple ninguna de las dos condiciones anteriores, no entra a ejecutar las instrucciones del ciclo.

17.6 Instrucción PARA *versus* instrucción MIENTRAS

Veamos cómo queda el algoritmo para generar e imprimir las tablas de multiplicar de un número dado, utilizando la instrucción PARA:

```
1.   Algoritmo tablaDeMultiplicarDeX
2.       Variables: x, r, i: numericaEntera
3.       INICIO
4.           LEA(x)
5.           PARA i DESDE 1 HASTA 5 CON_VARIACION +1
6.               r = x * i
7.               ESCRIBA(x, "*", i, "=", r)
8.           Fin(PARA)
9.       FIN
10.  Fin(tablaDeMultiplicarDeX)
```

Si comparamos este algoritmo con el que teníamos utilizando la instrucción MIENTRAS, podríamos decir que es más compacto.

Ahora, ¿cuándo utilizar la instrucción MIENTRAS y cuándo utilizar la instrucción PARA? La respuesta es más bien sencilla: cuando la terminación del ciclo esté controlada por el resultado de una operación es necesario utilizar MIENTRAS en vez de PARA. Por ejemplo, en la determinación de si un número **x** es primo o no, el ciclo MIENTRAS se controla con la condición **x** % **i** ≠ 0. Esta condición no se puede controlar con un ciclo PARA. La instrucción PARA es recomendable utilizarla cuando las instrucciones del ciclo se ejecutan un número fijo de veces, con una variación predefinida.

Resumen

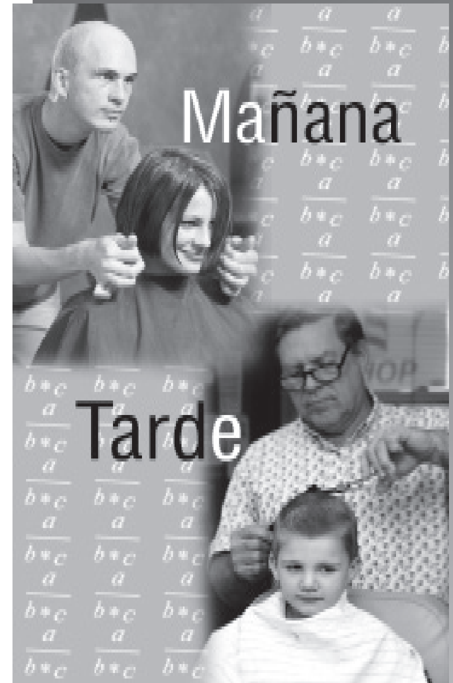
En este módulo presentamos una nueva instrucción de ciclo, la instrucción PARA, con la cual se pueden establecer ciclos de una manera más concisa cuando las características del ciclo así lo permiten.

Ejercicio propuesto

1. Trate de elaborar los mismos ejercicios de los módulos 14 y 15 utilizando sólo la instrucción PARA.

Módulo 18

Ciclos anidados con instrucción PARA



Introducción

Cuando tratamos la instrucción MIENTRAS vimos que hay situaciones en las cuales es necesario codificar instrucciones MIENTRAS dentro de instrucciones MIENTRAS. A esta forma de construcción la denominamos ciclos anidados. Con la instrucción PARA se tienen exactamente las mismas necesidades que con la instrucción MIENTRAS. En este módulo trataremos ejemplos concernientes a esta situación. Veremos cómo muchos de los algoritmos que se efectúan con ciclos anidados utilizando la instrucción MIENTRAS también se pueden elaborar utilizando la instrucción PARA.

Objetivos del módulo

1. Aplicar la instrucción PARA en algoritmos con ciclos anidados.

Preguntas básicas

1. ¿Cómo se define el factorial de un número entero?

Contenidos del módulo

18.1 Ejemplo 1
18.2 Ejemplo 2

«Si consideramos nuevamente el ejemplo de la barbería planteado en el módulo 17, pero esta vez con dos barberos que atienden en diferentes turnos, se pueden plantear ciclos anidados, con la instrucción PARA, de la siguiente forma: en la barbería se atiende desde las 7 a. m. hasta las 7 p. m.: desde las 7 a. m. y hasta la 1 p. m. atiende Antonio, y desde la 1 p.m. hasta las 7 p. m. atiende Aníbal. Los ciclos de atención de los barberos se hallan dentro del ciclo de atención en la barbería».

Vea en el botón **PARA** del mapa conceptual el video "Módulo 18. Instrucción PARA y los ciclos anidados".

18.1 Ejemplo 1

Consideremos el algoritmo para generar e imprimir las tablas de multiplicar desde uno hasta **n**, siendo **n** un valor leído. Cada tabla de multiplicar la generamos desde 1 hasta 10.

Nuestro algoritmo es:

```
1.  Algoritmo tablasDeMultiplicar(2)
2.      Variables: n, i, x, r: numericaEntera
3.      INICIO
4.          LEA(n)
5.          PARA x DESDE 1 HASTA n CON_VARIACION +1
6.              PARA i DESDE 1 HASTA 10 CON_VARIACION +1
7.                  r = x * i
8.                  ESCRIBA(x, "*", i, "=", r)
9.              Fin(PARA)
10.         Fin(PARA)
11.     FIN
12. Fin(tablasDeMultiplicar)
```

En la instrucción 4 se lee el valor de **n**.

En la instrucción 5 se plantea el ciclo externo, en el cual la variable **x** tomará valores desde 1 hasta **n**. Por cada valor de **x** se ejecuta el ciclo interno, que se inicia en la instrucción 6 y en el cual la variable **i** toma los valores desde 1 hasta 10.

Las instrucciones 7 y 8 son las instrucciones pertenecientes al ciclo interno.

La instrucción 9 delimita el alcance del ciclo interno. Al llegar a esta instrucción el programa controla que el valor de **i** no sea mayor que 10. Si el valor de **i** es menor o igual que 10, incrementa el valor de **i** en 1 y vuelve a ejecutar las instrucciones 7 y 8. Cuando el valor de **i** sea mayor que 10 termina la ejecución del ciclo interno y continúa ejecutando la instrucción 10.

En la instrucción 10 se controla que el valor de **x** sea menor o igual que **n**. Si el valor de **x** es menor o igual que **n**, incrementa la **x** en 1 y vuelve a ejecutar las instrucciones 6 a 9. Cuando el valor de **x** sea mayor que **n** termina la ejecución del ciclo externo y termina la ejecución del algoritmo.

18.2 Ejemplo 2

Consideremos un algoritmo que, aunque no trabaja con ciclos PARA anidados, es de gran utilidad en el área de matemáticas, y además es un ejemplo típico de la utilización de la instrucción PARA: elaborar un algoritmo que lea un entero positivo **n** y que calcule e imprima el factorial de **n**. El factorial de un entero positivo **n** se define como el producto de todos los enteros desde 1 hasta **n**.

Nuestro algoritmo, utilizando la instrucción PARA, es el siguiente:

```
1.  Algoritmo factorial
2.      Variables: n, i, f: numericaEntera
3.      INICIO
4.          LEA(n)
```

```
5.              f = 1
6.              PARA i DESDE 1 HASTA n CON_VARIACION +1
7.                  f = f * i
8.              Fin(PARA)
9.              ESCRIBA(n, f)
10.         FIN
11.   Fin(factorial)
```

Dejamos al lector la tarea de hacer prueba de escritorio al anterior algoritmo.

Resumen

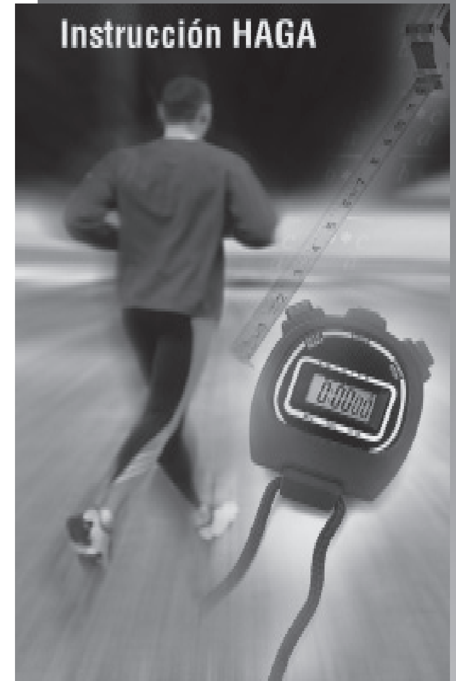
Al igual que con la instrucción MIENTRAS, con la instrucción PARA también se pueden construir ciclos anidados. En este módulo hemos presentado un ejercicio en el cual se usa esta propiedad.

Ejercicios propuestos

1. Trate de elaborar algoritmos, utilizando la instrucción PARA, con el fin de dar solución a los ejercicios propuestos en el módulo 16.

Módulo 19

Instrucción HAGA



Introducción

En los módulos anteriores se han tratado las instrucciones de ciclo MIENTRAS y PARA, que tienen la característica de que chequean la condición antes de ejecutar las instrucciones del ciclo. En este módulo se estudia la instrucción HAGA con la cual se tiene la posibilidad de ejecutar primero las instrucciones del ciclo y al final controlar la condición de repetición de las instrucciones.

Objetivos del módulo

1. Reconocer y aplicar la instrucción de ciclo HAGA.
2. Identificar cuándo utilizar la instrucción HAGA.
3. Codificar la instrucción HAGA como instrucción MIENTRAS.

Preguntas básicas

1. ¿Cuál es la principal característica de la instrucción HAGA?
2. ¿Cuándo se utiliza la instrucción HAGA?
3. ¿Cómo se convierte un ciclo HAGA en un ciclo MIENTRAS?

Contenidos del módulo

19.1 Forma general
19.2 Pertinencia de la instrucción HAGA
19.3 Ejemplo de uso
19.4 Conversión de la instrucción HAGA en instrucción MIENTRAS

«Hay situaciones en las cuales es necesario que las actividades del ciclo se ejecuten al menos una vez. Si consideramos una competencia atlética, la tarea de medir el tiempo implica que el cronómetro se accione en el preciso instante en que se da la partida. El ciclo se plantea así: mantenga accionado el cronómetro mientras no llegue el primer competidor».

Vea en el botón **HAGA** del mapa conceptual el video "Módulo 19. Instrucción HAGA".

19.1 Forma general

La forma general de la instrucción HAGA es:

```
HAGA
        Conjunto de instrucciones que se ejecutan mientras la condición sea
        verdadera.
MIENTRAS condición
```

Esta instrucción de ciclo tiene la característica de que se ejecuta al menos una vez, ya que la condición se evalúa después de que se han ejecutado las instrucciones del ciclo.

19.2 Pertinencia de la instrucción HAGA

Cuando se elaboran algoritmos es necesario, en algunas situaciones, poder ejecutar primero las instrucciones correspondientes a un ciclo y luego evaluar las condiciones que determinan si se vuelven a ejecutar las instrucciones del ciclo. Una de las situaciones en las que es pertinente utilizar la instrucción HAGA es en la elaboración de menús, en los cuales el usuario debe elegir una opción, ya que primero se deben presentar las posibles opciones para que el usuario escoja una, y luego evaluar la escogencia del usuario.

19.3 Ejemplo de uso

Elabore un algoritmo que presente por pantalla un menú con las siguientes opciones:

1. leer número.
2. calcular factorial.
3. determinar si es par.
4. terminar.

El usuario elige una opción y el programa opera de acuerdo a la opción que él eligió.

Veamos cómo es nuestro algoritmo:

```
1.   Algoritmo menu(1)
2.       Variables: n, f, i, op: numericaEntera
3.       INICIO
4.           HAGA
5.               ESCRIBA("1. leer número")
6.               ESCRIBA("2. calcular factorial")
7.               ESCRIBA("3. determinar si es par")
8.               ESCRIBA("4. terminar")
9.               ESCRIBA("elija opción")
10.              LEA(op)
11.              CASOS
12.                 :op == 1:   LEA(n)
13.                             SALTE
14.                 :op == 2:   f = 1
15.                     PARA i DESDE 1 HASTA n CON_VARIACION +1
16.                             f = f * i
17.                     Fin(PARA)
18.                     SALTE
19.                 :op == 3:   SI n % 2 == 0
20.                             ESCRIBA(n, "es par")
```

```
21.                                     DE_LO_CONTRARIO
22.                                               ESCRIBA(n, " es impar")
23                                      Fin(SI)
24.                                     SALTE
25.                        :op == 4:  SALTE
26.                         :OTRO_CASO:  ESCRIBA("opción inválida, teclee de
                                                            nuevo")
27.                    Fin(CASOS)
28.              MIENTRAS(op ≠ 4)
29.        FIN
30.   Fin(menu)
```

En la instrucción 2 definimos las variables con las cuales vamos a trabajar: **n** el número leído, **f** para almacenar el factorial, **i** para efectuar las operaciones de cálculo del factorial y **op** para almacenar la opción elegida por el usuario.

En la instrucción 4 hemos colocado la instrucción HAGA, la cual significa que todas las instrucciones que hay desde la 5 hasta la 27 se van a repetir mientras la condición escrita en la instrucción 27 sea verdadera, es decir, mientras **op** sea diferente de 4.

En las instrucciones 5 a 9 escribimos los mensajes de las diferentes opciones que tiene el usuario.

En la instrucción 10 se lee la opción elegida por el usuario y se almacena en la variable **op**.

En la instrucción 11 planteamos la instrucción CASOS, ya que necesitamos comparar el contenido de la variable **op** con varios valores: si **op** vale 1, simplemente se leerá un valor para **n**; si **op** vale 2, se calcula el factorial de **n** y se imprime dicho resultado; si **op** vale 3, averiguamos si **n** es par y producimos el mensaje apropiado; si **op** vale 4, simplemente instruimos la máquina para que no ejecute ninguna operación y vaya a la instrucción 28 para evaluar la condición de terminación del ciclo; y finalmente, si el valor de **op** no es ninguno de los anteriores, producimos el mensaje de que la opción entrada por el usuario no es válida. En las instrucciones correspondientes a cada una de las opciones hemos colocado la instrucción SALTE, la cual hace que vaya a la instrucción 28 para evaluar la condición del ciclo.

En la instrucción 28 se evalúa la condición: si **op** vale 4, la condición es falsa y la ejecución continúa en la instrucción siguiente, es decir, la instrucción 29; cualquier otro valor diferente de 4 significa que la condición es verdadera y por consiguiente regresa a la instrucción 11 para volver a ejecutar las instrucciones del ciclo.

19.4 Conversión de la instrucción HAGA en instrucción MIENTRAS

En general, cualquier ciclo elaborado con la instrucción HAGA se puede construir usando la instrucción MIENTRAS, aunque para ello se necesita una variable adicional.

Llamemos a esta variable adicional **sw**, la cual puede ser perfectamente un bit y es la variable con la que controlaremos el ciclo MIENTRAS. Inicialmente a esta variable le asignamos el valor de cero y planteamos el ciclo MIENTRAS para que se ejecute mientras **sw** sea igual a cero. El valor del **sw** sólo lo pondremos en 1 cuando la opción elegida por el usuario sea 4.

Veamos cómo queda nuestro algoritmo utilizando la instrucción MIENTRAS en vez de la instrucción HAGA.

```
1.    Algoritmo menu(2)
2.          Variables: n, f, i, op, sw: numericaEntera
3.          INICIO
4.                sw = 0
5.                MIENTRAS  sw == 0
6.                      ESCRIBA("1. leer número")
7.                      ESCRIBA("2. calcular factorial")
8.                      ESCRIBA("3. determinar si es par")
9.                      ESCRIBA("4. terminar")
10.                     ESCRIBA("elija opción")
11.                     LEA(op)
12.                     CASOS
13.                           :op == 1:   LEA(n)
14.                                       SALTE
15.                           :op == 2:   f = 1
16.                               PARA  i  DESDE 1 HASTA  n  CON_VARIACION  +1
17.                                           f = f * i
18.                               Fin(PARA)
19.                                       SALTE
20.                           :op == 3:   SI  n % 2  ==  0
21.                                           ESCRIBA(n, " es par")
22.                                       DE_LO_CONTRARIO
23.                                           ESCRIBA(n, " es impar")
24.                                       Fin(SI)
25.                                       SALTE
26.                           :op == 4:   sw = 1
27.                           :OTRO_CASO:  ESCRIBA("opción inválida, teclee de
                                                         nuevo")
28.                     Fin(CASOS)
29.               Fin(MIENTRAS)
30.         FIN
31.   Fin(menu)
```

Resumen

En este módulo hemos presentado la tercera instrucción de ciclo, la instrucción HAGA, que tiene como principal característica que primero ejecuta las instrucciones del ciclo y luego chequea la condición de repetición de las instrucciones.

Ejercicio propuesto

1. Trate de elaborar algoritmos correspondientes a los ejercicios propuestos de los módulos 15 y 16 utilizando únicamente la instrucción de ciclo HAGA.

Capítulo 5 Subprogramas

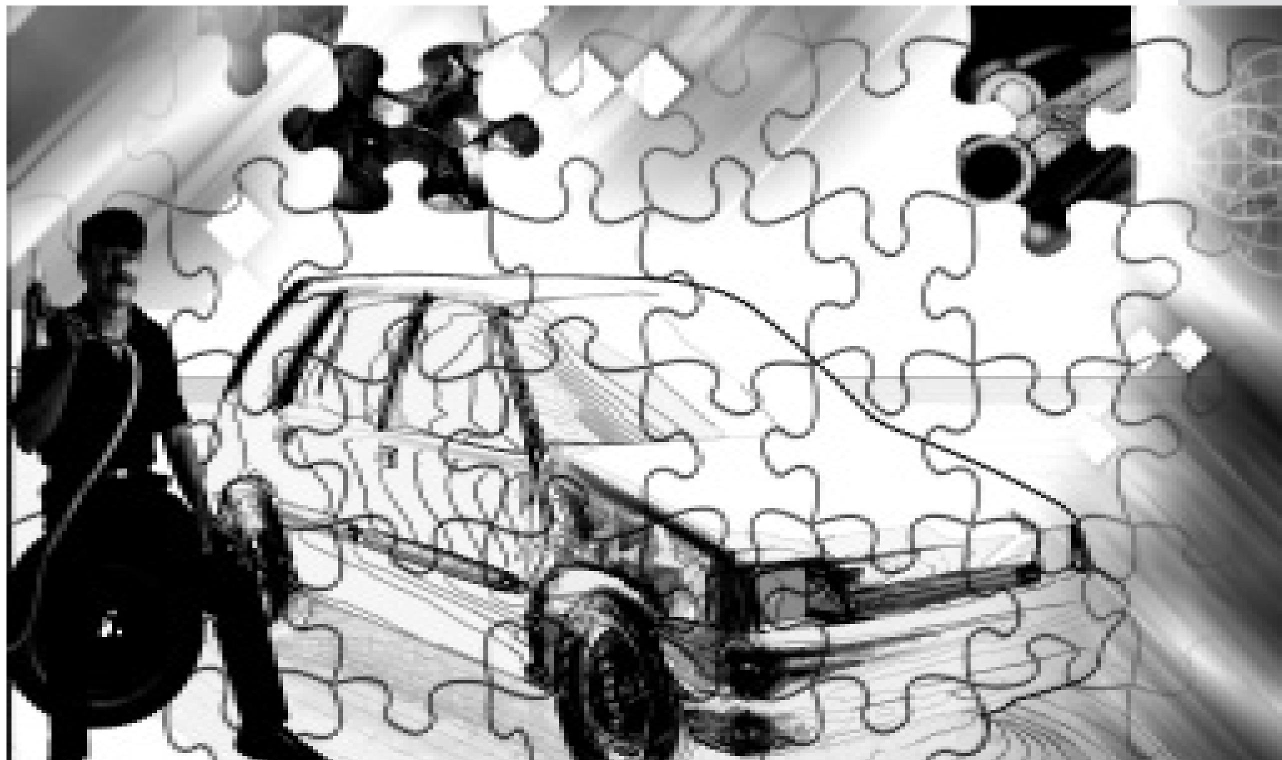
Contenido breve

Módulo 20
Concepto e identificación
Módulo 21
Clases y aplicación
Módulo 22
Parámetros y variables
Módulo 23
Ejemplos de uso
Módulo 24
Ejemplos de uso mejorados

La solución de problemas complejos se facilita considerablemente si se dividen en problemas más pequeños. La solución de estos problemas pequeños se realiza con pequeños algoritmos que se denominan subprogramas y cuya ejecución depende de un programa principal. Los subprogramas son unidades de programa que están diseñados para ejecutar una tarea específica. Estos subprogramas pueden ser funciones o procedimientos, los cuales se escriben sólo una vez, pero pueden ser referenciados en diferentes partes del programa principal evitando así la duplicidad de instrucciones en un mismo programa.

En este capítulo trataremos todo lo referente a funciones y procedimientos, junto con los conceptos de parámetros por valor, parámetros por referencia, variables locales y variables globales.

La construcción de grandes obras siempre es el acople de pequeñas obras.



Módulo 20

Concepto e identificación

Introducción

En el desarrollo de soluciones, utilizando el computador como herramienta, se presentan problemas muy complejos. Una de las principales técnicas para solucionar estos problemas es identificar las diferentes tareas que hay que ejecutar y resolver cada una de ellas en forma independiente, para luego construir la solución ensamblando como un todo las tareas que se solucionaron independientemente. El algoritmo correspondiente a cada una de esas tareas se denomina subprograma. En este módulo trataremos el concepto de subprograma y un ejemplo que lo ilustre.

Objetivos del módulo

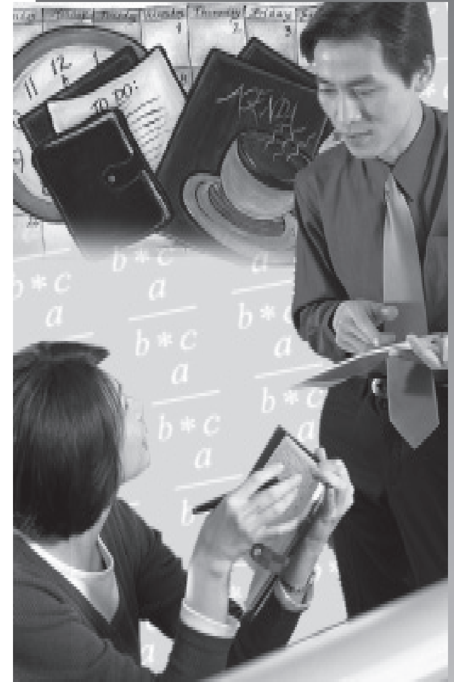
1. Reconocer el concepto de subprograma.
2. Identificar cuándo utilizar subprogramas.

Preguntas básicas

1. ¿Qué es un subprograma?
2. ¿Cómo identifico que hay que elaborar subprogramas?
3. ¿Dónde se invocan los subprogramas?

Contenidos del módulo

20.1 Definición
20.2 Ejemplo de uso



«Cuando se va a ejecutar un proyecto de gran tamaño, una de las formas de hacerlo es identificando las diferentes tareas que hay que efectuar para llevarlo a feliz término. Teniendo identificada cada una de las tareas, podemos asignárselas a diferentes personas y luego un coordinador se encarga de ensamblar cada uno de los resultados para obtener el producto final».

Vea en el botón **Procedimental** del mapa conceptual el video "Módulo 20. Subprogramas".

20.1 Definición

Los subprogramas son algoritmos que se elaboran en forma independiente y que tienen la característica de que se pueden invocar desde cualquier programa cuando se necesiten. El objetivo principal de utilizar subprogramas es elaborar algoritmos más cortos, menos complejos, más legibles y más eficientes.

20.2 Ejemplo de uso

En el tema de análisis combinatorio (conteo) es importante determinar el número de combinaciones que se pueden construir con **n** elementos, tomados en grupos de a **r** elementos. Por ejemplo, si tenemos tres elementos **a**, **b** y **c**, y queremos formar combinaciones con dos de esos tres elementos, las posibles combinaciones son **ab**, **ac** y **bc**.

El total de combinaciones de tres elementos (n == 3) tomando de a dos elementos (r == 2) es tres. Si el número de elementos es cuatro (n == 4), **a**, **b**, **c** y **d,** las combinaciones posibles tomando de a dos elementos (r == 2) son **ab**, **ac**, **ad**, **bc**, **bd** y **cd**, es decir, seis posibles combinaciones.

Para determinar el total de combinaciones de **n** elementos tomados en grupos de a **r** se tiene una fórmula matemática que se escribe así:

$$^{n}C_{r} = \frac{n!}{r!(n-r)!}$$

la cual se lee: el total de combinaciones de **n** elementos tomados en grupos de a **r** elementos es el factorial de **n** dividido por el producto del factorial de **r** con el factorial de **n – r**.

Si nuestro objetivo es elaborar un algoritmo en el cual se lean los datos de **n** y **r**, y calcular el número de combinaciones que se pueden construir, debemos definir una variable para el factorial de **n**, otra variable para el factorial de **r** y otra para el factorial de **n – r**. Llamemos a estas variables **fn**, **fr** y **fnr**.

Construyamos un algoritmo para ejecutar esta tarea: leer dos datos **n** y **r** y calcular e imprimir el total de combinaciones que se pueden construir con **n** elementos tomados en grupos de a **r** elementos:

```
1.   Algoritmo combinaciones(1)
2.        Variables: n, r, fn, fr, fnr, tc: numericaEntera
3.        INICIO
4.             LEA(n, r)
5.             fn = 1
6.             PARA i DESDE 1 HASTA n CON_VARIACION +1
7.                  fn = fn * i
8.             Fin(PARA)
9.             fr = 1
10.            PARA i DESDE 1 HASTA r CON_VARIACION +1
11.                 fr = fr * i
12.            Fin(PARA)
13.            fnr = 1
14.            PARA i DESDE 1 HASTA n – r  CON_VARIACION +1
15.                 fnr = fnr * i
16.            Fin(PARA)
```

```
17.            tc = fn / (fr * fnr)
18.            ESCRIBA(n, r, tc)
19.        FIN
20.    Fin(combinaciones)
```

En este algoritmo las instrucciones 5 a 8, las instrucciones 9 a 12 y las instrucciones 13 a 16 son exactamente las mismas; la única diferencia es que actúan sobre diferentes datos.

Las instrucciones 5 a 8 trabajan con las variables **fn** y **n**, las instrucciones 9 a 12 con las variables **fr** y **r**, y las instrucciones 13 a 16 con las variables **fnr** y el resultado de **n – r**.

Esta situación, en la cual tenemos un grupo de instrucciones que se repiten en diferentes partes del programa con datos diferentes, amerita una herramienta que nos permita obviar estas repeticiones.

Veamos cómo pudimos haber escrito nuestro algoritmo si tuviéramos un subprograma que determine el factorial de un número entero cualquiera:

```
1.     Algoritmo combinaciones(2)
2.         Variables: n, r, fn, fr, fnr, tc: numericaEntera
3.         INICIO
4.             LEA(n, r)
5.             fn = factorial(n)
6.             fr = factorial(r)
7.             fnr = factorial(n – r)
8.             tc = fn / fr / fnr
9.             ESCRIBA(n, r, tc)
10.        FIN
11.    Fin(combinaciones)
```

En este nuevo algoritmo hemos reemplazado las instrucciones 5 a 8 por una sola instrucción:

fn = factorial(n)

las instrucciones 9 a 12 por una sola instrucción:

fr = factorial(r)

y las instrucciones 13 a 16 por una sola instrucción:

fnr = factorial(n – r)

Como se podrá observar, el algoritmo (2) de *combinaciones* es más compacto y más legible que el algoritmo (1) de *combinaciones*.

Hemos usado un subprograma que llamamos *factorial*, el cual calcula y retorna el factorial de un número entero que se envía a este subprograma.

Veamos cómo es este subprograma:

```
1.   Entero factorial(m)
2.        Variables: i, f: numericaEntera
3.        INICIO
4.             f = 1
5.             PARA i DESDE 1 HASTA m CON_VARIACION +1
6.                  f = f * i
7.             Fin(PARA)
8.             retorne(f)
9.        FIN
10.  Fin(factorial)
```

En la instrucción 1 definimos el nombre del subprograma y lo precedemos con la palabra entero, la cual significa que este subprograma retorna un valor numérico de tipo entero.

En la misma instrucción 1 definimos una variable **m**, la cual será la variable que recibe el dato con el que trabajará el subprograma.

En la instrucción 2 definimos las variables propias que necesita el subprograma para poder efectuar los cálculos que se necesitan.

Las instrucciones 4 a 7 son las instrucciones propias para calcular el factorial de un número **m**.

En la instrucción 8 se retorna el valor que fue calculado para el dato **m**.

Nuestro objetivo, de aquí en adelante, será elaborar subprogramas que ejecuten tareas básicas, y con base en ellos elaborar algoritmos más complejos sin tener que escribir algoritmos demasiado extensos.

Resumen

Hemos introducido en este módulo el concepto de subprograma, una poderosa herramienta que nos permite construir algoritmos no sólo de una manera más sencilla, sino reutilizables.

Ejercicios propuestos

1. Considerando el subprograma *factorial* elaborado en el módulo, haga prueba de escritorio al siguiente algoritmo indicando claramente qué imprime.

```
1.   Algoritmo prueba
2.        Variables: a, b, c, d, e, f, g, h: numericaEntera
3.        INICIO
4.             a = 5
5.             b = 3
6.             c = 4
7.             f = factorial(b)
8.             g = factorial(a)
9.             h = factorial(c)
10.            ESCRIBA(f, g, h)
11.            d = f * g / h
```

```
12.         e = d * a
13.         ESCRIBA(d, e)
14.     FIN
15. Fin(prueba)
```

2. Escriba un algoritmo que ejecute lo mismo que el algoritmo anterior pero considerando que no dispone de un subprograma que calcule el factorial de un número.

Módulo 21

Clases y aplicación

Introducción

Cuando una tarea grande se subdivide en pequeñas tareas, cada una de ellas tendrá una característica especial: retornar un dato o simplemente ejecutar la tarea. Según esta característica se clasifican los subprogramas como funciones o como subprogramas tipo void. En este módulo tratamos esta problemática y definimos la clasificación y el uso de cada subprograma de acuerdo a la característica que tenga.

Objetivos del módulo

1. Identificar los diferentes tipos de subprogramas.
2. Reconocer cuándo aplicar cada tipo de subprograma.

Preguntas básicas

1. ¿Qué es una función?
2. ¿Qué es un subprograma tipo void?
3. ¿Cuándo se utiliza una función?
4. ¿Cuándo se utiliza un subprograma tipo void?

Contenidos del módulo

21.1 Prueba de escritorio al algoritmo del módulo 20
21.2 Clasificación de subprogramas
21.3 Ejemplo de subprograma tipo void

«Cuando se le asigna una tarea a una persona pueden suceder dos cosas: que me retorne un resultado o que simplemente ejecute la tarea. Un cirujano ejecuta una tarea en la cual no retorna ningún producto, pero un ebanista sí lo hace: retorna un comedor, una sala, etc.».

Vea en el botón **Procedimental** del mapa conceptual el video "Módulo 21. Subprogramas: parte 2".

21.1 Prueba de escritorio al algoritmo del módulo 20

En el módulo anterior hicimos un algoritmo utilizando el subprograma *factorial*. Nuestro algoritmo fue:

```
1.   Algoritmo combinaciones(2)
2.       Variables: n, r, fn, fr, fnr, tc: numericaEntera
3.       INICIO
4.           LEA(n, r)
5.           fn = factorial(n)
6.           fr = factorial(r)
7.           fnr = factorial(n – r)
8.           tc = fn / fr / fnr
9.           ESCRIBA(“n = “, n, “r = “, r, “ total combinaciones = “, tc)
10.      FIN
11.  Fin(combinaciones)
```

En las instrucciones 5, 6 y 7 invocamos un subprograma llamado *factorial* y le enviamos un dato diferente en cada instrucción. El subprograma *factorial* calcula el factorial del dato enviado y retorna dicho resultado.

```
1.   entero factorial(m)
2.       Variables: f, i: numericaEntera
3.       INICIO
4.           f = 1
5.           PARA i DESDE 1 HASTA m CON_VARIACION +1
6.               f = f * i
7.           Fin(PARA)
8.           retorne(f)
9.       FIN
10.  Fin(factorial)
```

Veamos cómo funciona nuestro programa *combinaciones* al ejecutarlo con datos 5 y 3.

Al ejecutar la instrucción 4 los datos 5 y 3 quedan almacenados en las variables **n** y **r** respectivamente.

La ejecución de la instrucción 5 consiste en invocar el subprograma que hemos llamado *factorial* y le enviamos como dato de trabajo el contenido de la variable **n**, es decir, 5. Como resultado de ejecutar esta instrucción la variable **fn** queda valiendo 120.

La instrucción 6 también invoca el subprograma *factorial*, y le enviamos como dato el contenido de la variable **r**, es decir, 3. Como resultado de ejecutar esta instrucción la variable **fr** queda valiendo 6.

La instrucción 7 también invoca el subprograma *factorial*, y le enviamos como dato el resultado de restarle **r** a **n**, el cual es 2. Como resultado de ejecutar esta instrucción la variable **fnr** queda valiendo 2.

En la instrucción 8 se calcula el total de combinaciones **tc**, el cual será el resultado de dividir 120 por 6 y luego por 2, obteniendo como resultado 10.

La instrucción 9 escribe los datos leídos y el total de combinaciones correspondientes, es decir, imprime:

n = 5 r = 3 total combinaciones = 10

21.2 Clasificación de subprogramas

Los subprogramas que retornan un valor se denominan *funciones*. Hay otro tipo de subprogramas que no retornan valores, sino que sólo ejecutan tareas. Estos subprogramas, que sólo ejecutan una tarea, se denominan tipo *void*.

21.3 Ejemplo de subprograma tipo void

Es muy común, en muchas actividades, poner en los informes encabezados que siempre tendrán los mismos títulos, y cuyos datos son siempre los mismos o la variación es mínima.

Para no tener que escribir en cada algoritmo las mismas instrucciones de escritura podemos elaborar un subprograma tipo void que efectúe esta tarea. Ejemplo: supongamos que los títulos son:

UNIVERSIDAD DE ANTIOQUIA
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA TELEMÁTICA
PROGRAMA Ude@
ALGORITMO: <u>nombre del algoritmo</u>

en donde la parte subrayada corresponde a un dato que se enviará como parámetro.

Veamos cómo sería nuestro algoritmo:

```
1.  Void titulos(nombre)
2.      INICIO
3.          ESCRIBA("UNIVERSIDAD DE ANTIOQUIA")
4.          ESCRIBA("FACULTAD DE INGENIERÍA")
5.          ESCRIBA("DEPARTAMENTO DE INGENIERÍA TELEMÁTICA")
6.          ESCRIBA("PROGRAMA Ude@")
7.          ESCRIBA("ALGORITMO: ", nombre)
8.      FIN
9.  Fin(titulos)
```

En la primera instrucción definimos el nombre del subprograma, el cual llamamos *títulos*, y le definimos un parámetro, llamado **nombre**, en el que se recibe el nombre del algoritmo cuyo título se desea imprimir. Ese dato que se recibe como parámetro lo imprimimos en la instrucción 7.

Las instrucciones 3, 4, 5 y 6 simplemente escriben los mensajes correspondientes a los títulos que se desean imprimir.

Teniendo estos subprogramas, *factorial(m)* y *títulos(nombre)*, podremos escribir nuestro algoritmo *combinaciones* así:

```
1.  Algoritmo combinaciones(3)
2.      Variables: n, r, fn, fr, fnr, tc: numericaEntera
3.      INICIO
4.          LEA(n, r)
```

```
5.            titulos("combinaciones")
6.            fn = factorial(n)
7.            fr = factorial(r)
8.            fnr = factorial(n – r)
9.            tc = fn / fr / fnr
10.           ESCRIBA("n = ", n, "r = ", r, " total combinaciones = ", tc)
11.       FIN
12.   Fin(combinaciones)
```

Al ejecutar este algoritmo con los mismos datos, 5 y 3, el resultado de la ejecución es:

```
UNIVERSIDAD DE ANTIOQUIA
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA TELEMÁTICA
PROGRAMA Ude@
ALGORITMO: combinaciones

n = 5    r = 3   total combinaciones = 10
```

Resumen

En este módulo hemos presentado las diferentes clases de subprogramas que están definidos: los subprogramas tipo void, que ejecutan una tarea, y los subprogramas con tipo, que retornan un valor.

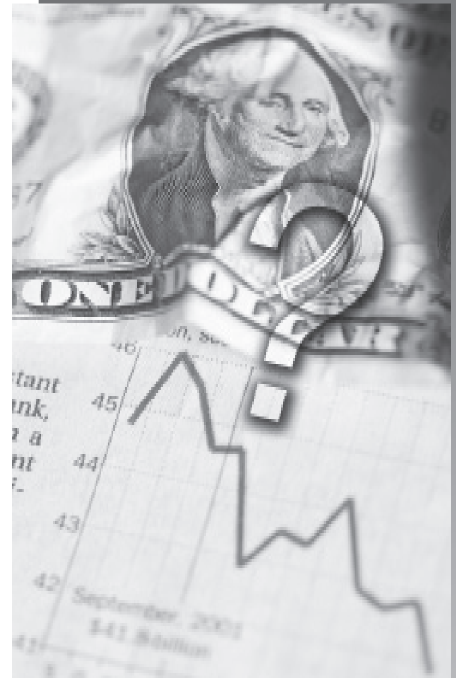
Ejercicios propuestos

1. Describa la diferencia entre una función y un subprograma tipo void.

2. Reescriba el algoritmo *combinaciones(3)* considerando que no dispone de una herramienta subprograma.

Módulo 22

Parámetros y variables



Introducción

Cuando se construyen subprogramas, éstos requieren datos para poder ejecutarlos. Estos datos hay que suministrárselos al subprograma usando otras variables. Estas variables reciben el nombre de parámetros. Adicionalmente, el subprograma requiere manejar otra serie de datos, bien sea definidos dentro del subprograma o tomados del programa principal. Todo lo anterior nos lleva a definir lo que son los conceptos de parámetros por valor, parámetros por referencia, variables locales y variables globales, que serán los temas que se tratarán en este módulo.

Objetivos del módulo

1. Identificar la forma como se definen los subprogramas.
2. Reconocer variables locales y variables globales.
3. Reconocer parámetros por valor y parámetros por referencia.

Preguntas básicas

1. ¿Cuál es la forma general de una función?
2. ¿Cuál es la forma general de un subprograma tipo void?
3. ¿Qué es una variable local?
4. ¿Qué es una variable global?
5. ¿En qué se diferencia un parámetro por valor de un parámetro por referencia?

Contenidos del módulo

22.1 Forma general de funciones y de subprogramas tipo void
22.2 Variables locales y variables globales
22.3 Parámetros de un subprograma

«Dinero en acciones o dinero en banco. Si invierto mi dinero en acciones, el total de mi capital varía dependiendo de si las acciones suben o bajan de precio, es decir, cualquier fluctuación en el valor de las acciones afecta la cantidad de dinero que deposité en ellas; pero si lo deposito en un banco, el total de mi dinero será el mismo, independientemente de si el banco gana o pierde. Mi dinero en acciones es dinero por referencia, y mi dinero en bancos es dinero por valor».

Vea en el botón **Procedimental** del mapa conceptual el video "Módulo 22. Subprogramas: parte 3".

22.1 Forma general de funciones y de subprogramas tipo void

Hemos definido dos tipos de subprogramas:

- Funciones: retornan un valor
- Void: ejecutan una tarea

La forma general de una función es:

Tipo de dato que retorna nombre_de_la_función(datos necesarios)

```
Variables:  v1, v2, v3, ...vn
INICIO
    ⋮
    retorne (vi)
FIN
```

Fin (nombre_de_la_función)

La forma de un subprograma void es:

Void nombre_del_sub_programa(datos necesarios)

```
Variables:  v1, v2, v3, ...vn
INICIO
    ⋮
FIN
```

Fin (nombre_del_sub_programa)

En la función, el tipo de dato que retorna puede ser entero, real, alfanumérico, etc. En nuestro ejemplo del factorial, el tipo que definimos para la función fue entero, ya que el dato que retorna es un número entero.

El llamado de la función debe estar en una instrucción de asignación. El llamado de un subprograma tipo void es simplemente una instrucción en la cual se hace referencia al nombre del subprograma, con sus respectivos parámetros.

22.2 Variables locales y variables globales

Dentro de un subprograma se pueden definir nuevas variables. Las variables que se definen dentro del subprograma se denominan *variables locales* y sólo existirán mientras se esté ejecutando el subprograma; cuando termina la ejecución del subprograma dichas variables desaparecen.

Si dentro de un subprograma se utilizan variables definidas por fuera del subprograma, éstas se llaman *variables globales*.

Los datos necesarios para la ejecución de un subprograma se denominan *parámetros formales*.

22.3 Parámetros de un subprograma

Los parámetros, por lo general, son datos de entrada al subprograma, aunque también puede haber parámetros de entrada y salida, o sólo de salida. Los parámetros de entrada se denominan *parámetros por valor*. Los parámetros de entrada y salida, o sólo de salida, se denominan *parámetros por referencia*. Cuando se define un subprograma hay que especificar cuáles parámetros son por valor y cuáles son por referencia.

Elaboremos un ejemplo para mostrar la diferencia entre parámetros por valor y parámetros por referencia:

```
1.  Algoritmo parametros
2.      Variables: a, b, c: numericaEntera
3.      INICIO
4.          LEA(a, b, c)
5.          demo(a, b, c)
6.          ESCRIBA(a, b, c)
7.      FIN
8.  Fin(parametros)
```

```
1.  Void demo(por valor: x, y; por referencia: z)
2.      INICIO
3.          x = x + 3
4.          y = y * x
5.          z = x + y
6.          ESCRIBA(x, y, z)
7.      FIN
8.  Fin(demo)
```

En el algoritmo que hemos llamado *parámetros*, se definen tres variables enteras **a**, **b** y **c**, las cuales vamos a suponer que se hallan almacenadas en las posiciones 5, 6 y 7 de memoria, respectivamente (figura 22.1).

En el subprograma *demo* se definen tres parámetros **x**, **y**, **z**. Los parámetros **x** e **y** se definen por valor, **z** por referencia. Supongamos que estos tres parámetros **x**, **y** y **z** se almacenan en las posiciones de memoria 19, 20 y 21, respectivamente (figura 22.1).

Al ejecutar la instrucción 4 del subprograma *parámetros*, se leen tres datos para **a**, **b** y **c**, digamos 2, 8 y 9. Estos datos quedan almacenados en las posiciones de memoria 5, 6 y 7.

Al ejecutar la instrucción 5, la cual invoca el subprograma *demo*, sucede lo siguiente: **x** e **y** fueron definidos parámetros por valor, por lo tanto el contenido de las variables **a** y **b** se copia hacia las posiciones de memoria correspondientes a **x** e **y**; **z** fue definido por referencia, lo cual implica que lo que se copia en la posición de memoria **z** es la posición de memoria en la cual se halla almacenada la variable **c** (figura 22.1).

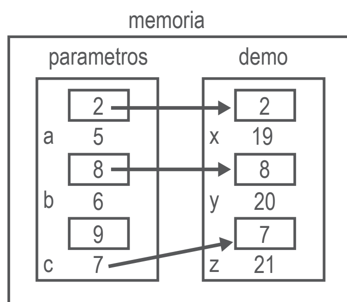


Figura 22.1. Estado de la memoria cuando se invoca el subprograma.

Al ejecutarse el subprograma *demo*, sucede lo siguiente: en la instrucción 3 se le suma 3 a **x**, y el resultado se almacena en la misma posición de memoria **x**; en la instrucción 4 se multiplica **y** por **x** y el resultado se almacena en la misma posición de memoria **y**; y en la instrucción 5 sumamos **x** con **y**, y almacenamos el resultado en **z**. ¿Pero qué es **z?** El parámetro **z** es una dirección de memoria: la dirección de memoria 7; por lo tanto el resultado de sumar el contenido de **x** con el contenido de **y** se almacena en la posición de memoria 7. Es como si desde dentro del subprograma hubiéramos tenido acceso a la variable **c** definida en el programa principal.

El resultado de efectuar estas operaciones se refleja en la figura 22.2.

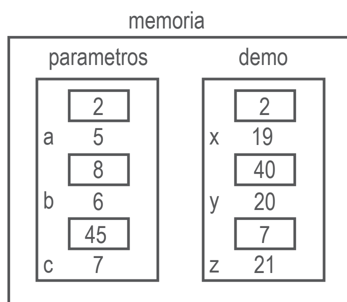


Figura 22.2. Estado de la memoria después de ejecutar el subprograma.

Al ejecutar la instrucción 6 del subprograma *demo*, imprime 5, 40 y 45.

Al ejecutar la instrucción 6 del subprograma *parametros*, imprime 2, 8 y 45.

Resumen

En este módulo hemos presentado los conceptos de parámetros por valor, parámetros por referencia, variables locales y variables globales, que son de gran importancia en la elaboración de subprogramas.

Ejercicio propuesto

1. Considere el siguiente subprograma:

```
1.  Entero misterio(m, i)      // m es parámetro por valor, i es parámetro por referencia
2.          t = m
3.          k = 0
4.          s = 10
5.          MIENTRAS  t  > 0  HAGA
6.                  j = t % s
7.                  k = k * s + j
8.                  t = t / s
9.                  i = i + 1
10.                 ESCRIBA(t, k , i)
11.         Fin(MIENTRAS)
12.         ESCRIBA(m, k, i)
13.         retorne(k)
14. Fin(misterio)
```

y el siguiente programa principal:

```
1. n = 675
2. u = 0
3. r = misterio(n, u)
4. ESCRIBA(n, r, u)
```

Haga prueba de escritorio detallada al programa principal y al subprograma indicando claramente qué imprime.

Módulo 23

Ejemplos de uso

Introducción

Hemos planteado que con la utilización de subprogramas los algoritmos son más legibles y fáciles de construir. En este módulo presentamos dos ejemplos en los cuales se muestra cómo se simplifica y se facilita la construcción de algoritmos usando subprogramas.

Objetivos del módulo

1. Aplicar los conocimientos adquiridos en cuanto al manejo de subprogramas.

Preguntas básicas

1. ¿Cuál entero representa verdadero?
2. ¿Cuál entero representa falso?
3. ¿Por qué es más legible un algoritmo que utiliza subprogramas?

Contenidos del módulo

23.1 Ejemplo 1
23.2 Ejemplo 2
23.3 Prueba de escritorio del ejemplo 2

«Para construir un auto se requiere la unión de varios componentes. Dichos componentes se mandan a construir, cada uno por aparte. Alguien construye el motor, otro las puertas, otro el sistema de frenos, y así sucesivamente. Teniendo todas las partes construidas procedemos a reunirlas en un todo para que el auto funcione como una sola unidad».

Vea en el botón **Procedimental** del mapa conceptual el video "Módulo 23. Subrogramas: parte 4".

23.1 Ejemplo 1

En el módulo 15 elaboramos un algoritmo con el cual se determinaba si un número entero es primo o no. Vamos a retomar este algoritmo y vamos a construir una función que tenga como parámetro un entero **n**, y que retorne verdadero o falso dependiendo de si el número **n** es primo o no.

En realidad nuestro subprograma retornará 1 si el número **n** es primo, o 0 si el número **n** no es primo, ya que por convención se tiene establecido que el 0 representa falso y el 1 verdad.

Nuestra función es:

```
1.  entero esPrimo(x)
2.      Variable: i: numericaEntera
3.      INICIO
4.          i = 2
5.          MIENTRAS  (i <= sqrt(x) ^ x % i <> 0)
6.              i = i + 1
7.          Fin(MIENTRAS)
8.          SI  x % i  == 0
9.              retorne(0)
10.         DE_LO_CONTRARIO
11.             retorne(1)
12.         Fin(SI)
13.     FIN
14. Fin(esPrimo)
```

La diferencia de este algoritmo con el elaborado en el módulo 15 es que en la instrucción siguiente al fin del MIENTRAS, en la cual se pregunta si **x % i** es igual a cero, la acción que se toma no es producir el mensaje acerca de si **x** es primo o no, sino retornar 0 o 1, indicando que **x** es primo o no.

23.2 Ejemplo 2

Teniendo construida esta función y el subprograma *títulos* que elaboramos en el módulo 21, vamos a elaborar un ejemplo más completo de utilización de subprogramas.

Elabore un algoritmo que lea un entero **n** y que haga lo siguiente:

1. Calcule e imprima el factorial de **n**.
2. Determine e imprima si **n** es primo o no.
3. Determine e imprima todos los primos desde 3 hasta **n**.

Llamemos *subprogramas* a este algoritmo:

```
1.  Algoritmo subprogramas(1)
2.      Variables: n, fn, i: numericaEntera
3.      INICIO
4.          LEA(n)
5.          titulos("factorial")
6.          fn = factorial(n)
7.          ESCRIBA("n = ", n, " factorial = ", fn)
8.          titulos("es primo o no")
```

```
9.          SI  esPrimo(n)
10.              ESCRIBA(“n = “, n, “ es primo”)
11.         DE_LO_CONTRARIO
12.              ESCRIBA(“n = “, n, “ no es primo”)
13.         Fin(SI)
14.         titulos(“todos los primos”)
15.         PARA  i  DESDE  3  HASTA  n  CON_VARIACION  +1
16.              SI esPrimo(i)
17.                   ESCRIBA(i)
18.              Fin(SI)
19.         Fin(PARA)
20     FIN
21.  Fin(subprogramas)
```

23.3 Prueba de escritorio del ejemplo 2

Veamos cómo se comporta nuestro algoritmo cuando leamos un dato de 14.

Al ejecutar la instrucción 4 la variable **n** queda valiendo 14.

Al ejecutar las instrucciones 5, 6 y 7 se produce la siguiente salida:

UNIVERSIDAD DE ANTIOQUIA
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA TELEMÁTICA
PROGRAMA Ude@
ALGORITMO: factorial

n = 14   factorial = 87178291200

Al ejecutar las instrucciones 8 a 13 se produce la siguiente salida:

UNIVERSIDAD DE ANTIOQUIA
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA TELEMÁTICA
PROGRAMA Ude@
ALGORITMO: es primo o no

n = 14 no es primo

Al ejecutar la instrucción 14 se produce la siguiente salida:

UNIVERSIDAD DE ANTIOQUIA
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA TELEMÁTICA
PROGRAMA Ude@
ALGORITMO: todos los primos

Y al ejecutar las instrucciones 15 a 18 imprime:

3, 5, 7, 11, 13

Es decir, todos los primos entre 3 y 14.

Observe que la diferencia en la ejecución de las instrucciones 5, 8 y 14 es sólo el mensaje que aparece a continuación del título ALGORITMO.

Es bueno dar alguna explicación con respecto a las instrucciones 9 y 16, que son la pregunta

SI esPrimo(n)

Esta instrucción, así como está, es equivalente a tener la instrucción

SI esPrimo(n) == 1

Recordemos que nuestro subprograma *esPrimo(n)* retorna un entero, el cual puede ser 0 o 1. Si el entero retornado es 1 significa que la situación que controló el algoritmo del subprograma es verdadera; si el valor retornado es 0 significa que la situación es falsa.

Resumen

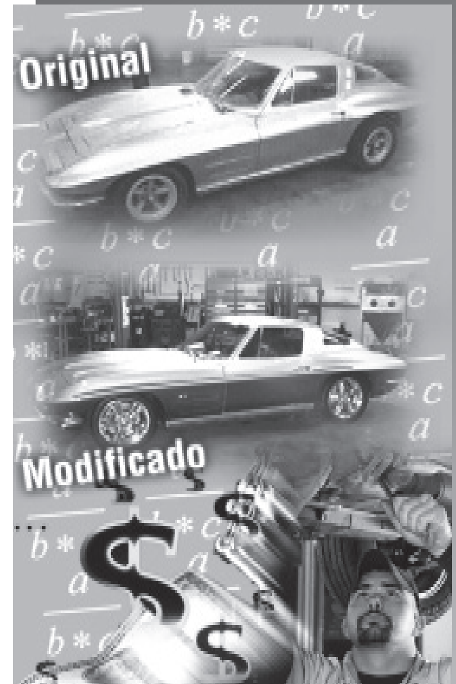
En este módulo hemos presentado ejercicios en los cuales aplicamos los conceptos de parámetros por valor y parámetros por referencia, como punto central de ellos.

Ejercicio propuesto

1. Reescriba el algoritmo *subprogramas(1)* del numeral 23.2 sin utilizar subprogramas.

Módulo 24

Ejemplos de uso mejorados



Introducción

En el módulo anterior se elaboró un algoritmo en el cual, al usar el subprograma *títulos*, éste imprime todos los mensajes contenidos en todas las instrucciones ES-CRIBA del subprograma. En este módulo presentamos una modificación a dicho subprograma de tal manera que cuando se invoque el subprograma *títulos* sólo imprima parte de ellos.

Objetivos del módulo

1. Aplicar más dinámicamente la utilización de subprogramas.

Preguntas básicas

1. ¿Cuál es el efecto de utilizar un parámetro por referencia?

Contenidos del módulo

24.1 Uso de parámetros por referencia en subprogramas tipo void

«Consideremos el caso de un automóvil que entra a un taller. Puede entrar a reparación o a remodelación. Si entra a reparación, su costo no se altera; en cambio, si entra a remodelación, todas las mejoras que se le hagan afectan el costo. En este último caso el costo del vehículo es un parámetro por referencia, ya que el trabajo que le hagan lo afecta».

Vea en el botón **Procedimental** del mapa conceptual el video "Módulo 24. Subprogramas: parte 5".

24.1 Uso de parámetros por referencia en subprogramas tipo void

En el ejemplo del módulo anterior, cuando usamos el subprograma *títulos(nombre)* del numeral 21.3, cada vez que se invoca escribe todos los títulos. Sin embargo, es más razonable que las primeras cuatro líneas sólo las escriba una vez; en las demás situaciones sólo se debe imprimir el título ALGORITMO y el nombre del algoritmo que se ejecutó.

Vamos a modificar nuestro subprograma *títulos(nombre)* para que se comporte de esta manera.

Introducimos un nuevo parámetro, el cual llamamos **sw** y que será un parámetro por referencia.

Cuando queramos que el subprograma *títulos(nombre, sw)* imprima todos los títulos invocamos el subprograma *títulos* enviándole al parámetro **sw** una variable con valor de 0.

Dentro del subprograma *títulos(nombre, sw)* se pregunta por el contenido de la variable **sw**. Si el **sw** vale 0 se imprimen todos los títulos y se asigna 1 a dicha variable, de tal manera que cuando se vuelva a invocar el subprograma *títulos(nombre, sw)* el contenido de la variable **sw** es 1, y por consiguiente no volverá a imprimir todos los títulos sino únicamente lo correspondiente a la instrucción 10.

Veamos cómo queda nuestro algoritmo *títulos*:

```
1.   void titulos(nombre, sw)    // sw parámetro por referencia
2.        INICIO
3.          SI sw == 0
4.               sw = 1
5.               ESCRIBA(“UNIVERSIDAD  DE  ANTIOQUIA”)
6.               ESCRIBA(“FACULTAD  DE  INGENIERÍA”)
7.               ESCRIBA(“DEPARTAMENTO  DE INGENIERÍA TELEMÁTICA”)
8.               ESCRIBA(“PROGRAMA Ude@”)
9.          Fin(SI)
10.         ESCRIBA(“ALGORITMO: “, nombre)
11.       FIN
12.  Fin(titulos)
```

Veamos cómo queda nuestro algoritmo *subprogramas*:

```
1.   Algoritmo subprogramas(2)
2.        Variables: n, fn, i, s:  numericaEntera
3.        INICIO
4.             s = 0
5.             LEA(n)
6.             titulos(“factorial”, s)
7.             fn = factorial(n)
8.             ESCRIBA(“n = “, n, “ factorial = “, fn)
9.             titulos(“es primo o no”, s)
10.            SI es_primo(n)
11.                 ESCRIBA(“n = “, n, “ es primo”)
12.            DE_LO_CONTRARIO
13.                 ESCRIBA(“n = “, n, “  no es primo”)
14.            Fin(SI)
15.            titulos(“todos los primos”, s)
16.            PARA  i  DESDE  3  HASTA  n  CON_VARIACION  +1
```

```
17.             SI es_primo(i)
18.                 ESCRIBA(i)
19.             Fin(SI)
20.         Fin(PARA)
21      FIN
22  Fin(subprogramas)
```

Definimos una nueva variable, la cual llamamos **s**.

En la instrucción 4 inicializamos la variable **s** con un valor de cero.

En la instrucción 6 se invoca el subprograma *títulos(nombre, sw)* y le enviamos **s** como parámetro.

Al ejecutarse el subprograma *títulos(nombre, sw)* la variable **s** queda valiendo 1; por lo tanto, cuando se ejecuten las instrucciones 9 y 15 sólo se imprimirá el mensaje correspondiente a la instrucción 10 del subprograma *títulos*.

Al ejecutar el algoritmo *subprogramas* con un dato leído de 14 se producirá el siguiente informe:

```
            UNIVERSIDAD  DE  ANTIOQUIA
             FACULTAD  DE  INGENIERÍA
   DEPARTAMENTO  DE  INGENIERÍA TELEMÁTICA
                PROGRAMA Ude@
              ALGORITMO: factorial

        n = 14    factorial = 87178291200

            ALGORITMO: es primo o no
              n = 14   no es primo

          ALGORITMO: todos los primos
                3, 5, 7, 11, 13
```

Resumen

En este módulo hemos presentado un ejemplo en el cual el uso de parámetros por referencia es más representativo de la potencialidad que ofrecen los parámetros por referencia.

Ejercicio propuesto

1. Elabore los algoritmos del módulo sin utilizar el parámetro por referencia **sw**, del subprograma *títulos(nombre, sw)*.

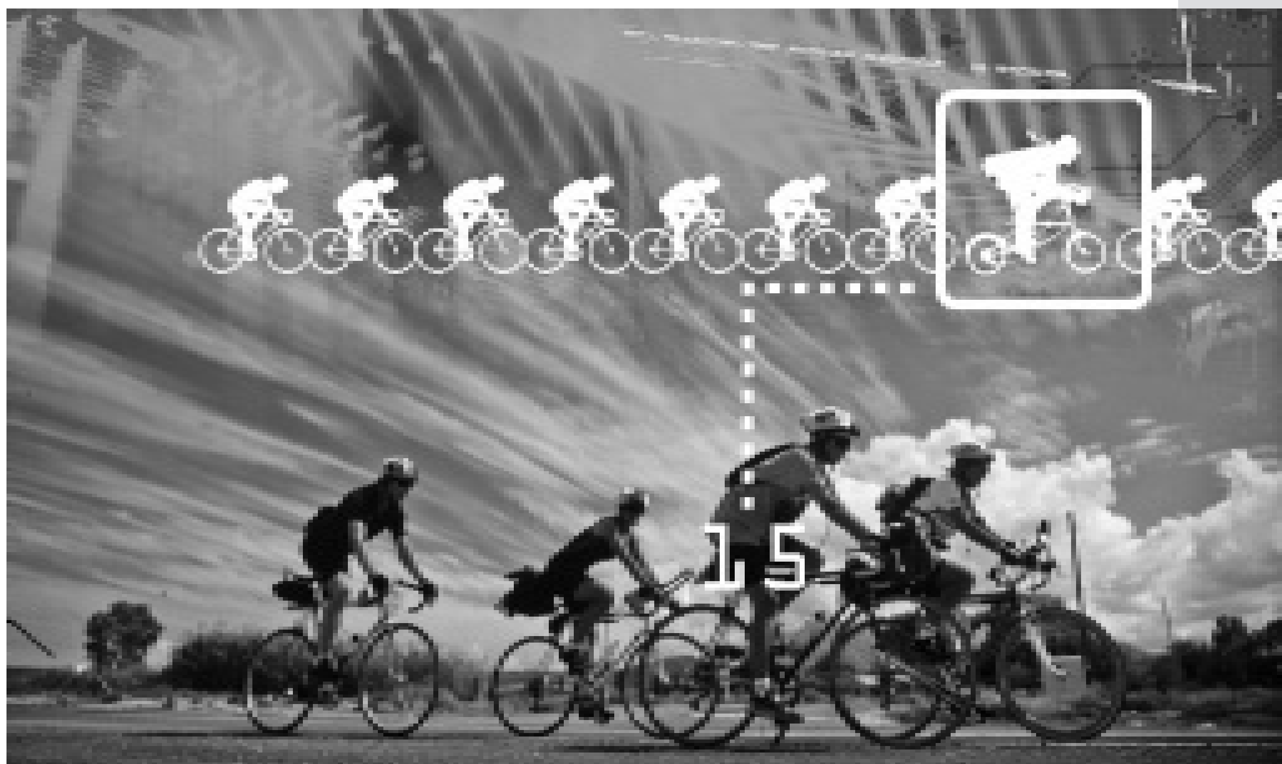
Capítulo 6 Vectores

Contenido breve

Módulo 25
Identificar el uso de un vector
Módulo 26
Definición y construcción
Módulo 27
Operaciones básicas: mayor dato, menor dato e intercambio de datos
Módulo 28
Operaciones básicas: proceso de inserción en un vector ordenado ascendentemente
Módulo 29
Operaciones básicas: proceso de borrado en un vector
Módulo 30
Operaciones básicas: búsqueda binaria y ordenamiento por selección
Módulo 31
Operaciones básicas: ordenamiento por burbuja

En los capítulos anteriores hemos tratado el concepto de variables simples, es decir, variables en las cuales sólo se almacena un dato. En muchas situaciones se necesita manipular colecciones de datos del mismo tipo, como por ejemplo las notas de los estudiantes de un curso, las ventas de un producto en diferentes almacenes, etc. El procesamiento de dichas colecciones utilizando variables simples es bastante impráctico y,

La organización y presentación lineal de un conjunto de datos es una necesidad en muchas situaciones de la vida diaria.

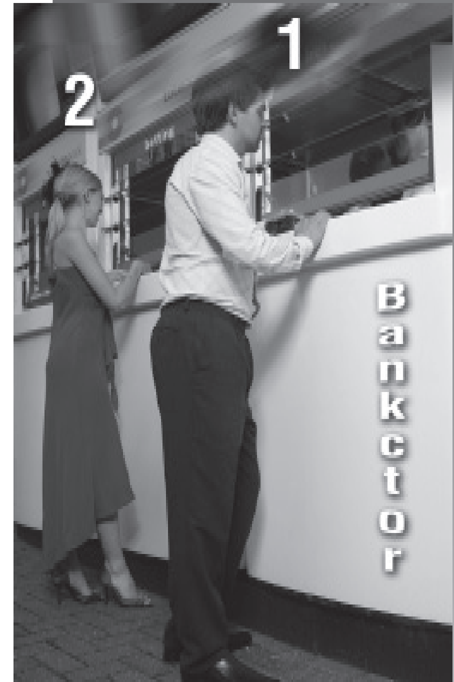


a veces, difícil, y por eso se han desarrollado estructuras que permiten manipularlas y procesarlas de una manera fácil y eficiente. Dichas estructuras se denominan arreglos.

Un arreglo es un conjunto de posiciones consecutivas de la memoria principal en las cuales se almacenan datos del mismo tipo y se puede acceder a ellos directamente mediante el uso de subíndices. En este capítulo estudiaremos arreglos de una dimensión, conocidos en el contexto informático como vectores.

Módulo 25

Identificar el uso de un vector



Introducción

En la práctica existen situaciones en las cuales se necesita manejar un gran volumen de datos del mismo tipo que implican una gran cantidad de variables, lo cual hace que la solución algorítmica sea muy rígida e impráctica. En este módulo presentamos una de estas situaciones y su solución, utilizando arreglos de una dimensión, los cuales se denominan vectores.

«Las personas que hacen fila para efectuar transacciones en la taquilla de un banco se pueden catalogar como un arreglo lineal de clientes. Cada uno de los clientes ocupa una posición en la fila, y podremos identificar a cada persona según la posición que ocupe en dicha fila: el cliente de la primera posición, el cliente de la segunda posición, el cliente de la tercera posición, etc. Ese arreglo lineal de clientes se conoce en el contexto de programación como un vector».

Objetivos del módulo

1. Reconocer la estructura estática de una dimensión para representar datos.

Preguntas básicas

1. ¿Qué es un vector?
2. ¿Cómo se referencia el dato de una posición?
3. ¿Qué es un subíndice?
4. ¿Cuándo se requiere el uso de un vector?

Contenidos del módulo

25.1 Descripción del problema
25.2 Solución al problema. Concepto de vector

Vea en el botón **Una dimensión** del mapa conceptual el video "Módulo 25. Concepto de arreglos".

25.1 Descripción del problema

Consideremos la situación de que se hizo un censo en la ciudad de Medellín y que se grabó un archivo en disco, llamado **'censo'**, el cual contiene la siguiente información en cada registro: municipio, dirección y número de personas. Cada registro contiene los datos correspondientes a cada vivienda visitada. Interesa procesar el archivo y calcular el total de personas que viven en Medellín.

Definimos las siguientes variables:

mpio: variable en la cual se almacena el código del municipio.
dir: variable en la cual se almacena la dirección de la vivienda visitada.
np: variable en la cual se almacena el número de personas de la vivienda visitada.
cmed: variable en la cual llevaremos el acumulado de las personas que viven en Medellín.

Un algoritmo para procesar el archivo '**censo**' es el siguiente:

```
1.    Algoritmo censoEnMedellin
2.          Variables: mpio, cmed, np: numericaEntera
3.                        dir: alfanumérica
4.          INICIO
5.                cmed = 0
6.                ABRA(censo)
7.                MIENTRAS not  EOF(censo)
8.                      LEA(censo: mpio, dir, np)
9.                      cmed = cmed + np
10.               Fin(MIENTRAS)
11.               CIERRE(censo)
12.               ESCRIBA(cmed)
13.         FIN
14.   Fin(censoEnMedellin)
```

Si el censo se hace en dos municipios (Medellín y Bello), y los códigos asignados a los municipios son:

mpio: { 1. Medellín: cmed
        2. Bello: cbel

un algoritmo para procesar el archivo '**censo**' y calcular e imprimir el total de habitantes de cada municipio es:

```
1.    Algoritmo censoEnMedellinYBello
2.          VARIABLES: mpio, cmed, cbel, np:  numericaEntera
3.                            dir: alfanuméricas
4.          INICIO
5.                ABRA(censo)
6.                cmed = 0
7.                cbel = 0
8.                MIENTRAS  not  EOF(censo)
9.                      LEA(censo: mpio, dir, np)
10.                     SI mpio == 1
11.                           cmed = cmed + np
12.                     DE_LO_CONTRARIO
```

```
13.                     cbel = cbel + np
14.                 Fin(SI)
15.             Fin(MIENTRAS)
16.             ESCRIBA(“habitantes Medellín: “, cmed, “ habitantes Bello: “, cbel)
17.             CIERRE(censo)
18.         FIN
19.     Fin(censoEnMedellinYBello)
```

Observe que en este algoritmo ya necesitamos dos acumuladores: uno para el total de habitantes en Medellín (**cmed**) y otro para el total de habitantes de Bello (**cbel**).

Si el censo hubiera sido para Medellín, Bello, Itagüí y Envigado se le asigna un código a cada municipio:

```
        ⎧1. Medellín: cmed
mpio:   ⎨2. Bello: cbel
        ⎨3. Itagüí: cita
        ⎩4. Envigado: cenv
```

Nuestro algoritmo es:

```
1.      Algoritmo censoEnMedellinYSusVecinos
2.          variables: mpio, cmed, cbel, cita, cenv, np: numericaEntera
3.                  dir: alfanuméricas
4.          INICIO
5.              ABRA(censo)
6.              cmed = 0
7.              cbel = 0
8.              cita = 0
9.              cenv = 0
10.             MIENTRAS  not EOF(censo)
11.                 LEA(censo: mpio, dir, np)
12.                 CASOS
13.                     :mpio == 1:     cmed = cmed + np
14.                                     SALTE
15.                     :mpio == 2:     cbel = cbel + np
16.                                     SALTE
17.                     :mpio == 3:     cita = cita + np
18.                                     SALTE
19.                     :mpio == 4:     cenv = cenv + np
20.                                     SALTE
21.                 Fin(CASOS)
22.             Fin(MIENTRAS)
23.             CIERRE(censo)
24.             ESCRIBA(“habitantes Medellín: “, cmed, “ habitantes Bello: “, cbel)
25.             ESCRIBA(“habitantes Itagüí: “, cita, “ habitantes Envigado: “, cenv)
26.         FIN
27.     Fin(censoEnMedellinYSusVecinos)
```

Observe que en este nuevo algoritmo ya se necesitan cuatro acumuladores: uno por cada municipio que se procese.

Si el censo hubiera sido para los 125 municipios del departamento, se requerirían 125 acumuladores: uno para cada municipio. Tendríamos que manejar 125 variables, nuestra instrucción CASOS sería muy extensa y el algoritmo sería completamente impráctico, ya que cada vez que varíe el número de municipios debemos modificar nuestro algoritmo.

25.2 Solución al problema. Concepto de vector

Presentaremos una solución alterna utilizando una estructura arreglo en la cual definimos un área de memoria con cierto número de posiciones, e identificamos cada posición con un número entero.

Veamos dicha estructura:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
acmpio	3	1	6	28	9	4	5	7	6	3	2	7	5	8

Nuestra estructura la hemos llamado **acmpio** y tiene una capacidad de catorce elementos. Para referirnos a algún elemento lo hacemos con el nombre de la estructura y un subíndice que indique la posición de dicho elemento.

Si nuestra estructura tiene los datos que se muestran y queremos imprimir el dato que se halla en la posición 4 escribimos:

```
ESCRIBA(acmpio[4])
```

y el resultado de ejecutar esta instrucción es que escribe el número 28.

Dicha estructura se conoce como un *arreglo de una dimensión* y en el contexto de algoritmos se denomina *vector*.

En general, para referirnos al contenido de una posición lo hacemos con el nombre del vector y un entero que se refiere a la posición. Dicho entero lo escribimos entre corchetes y lo seguiremos llamando *subíndice*.

Para el ejemplo que tenemos, si escribimos las siguientes instrucciones:

```
ESCRIBA("i  municipio")
PARA i DESDE 3 HASTA 6 CON_VARIACION +1
        ESCRIBA(i, acmpio[i])
Fin(PARA)
```

al ejecutarlas imprimirá:

```
i     municipio
3          6
4         28
5          9
6          4
```

Resumen

En este módulo hemos introducido el concepto de vector, una estructura estática de gran utilidad en la representación de grandes volúmenes de datos del mismo tipo.

Ejercicios propuestos

1. Dado el siguiente archivo llamado '**censo**':

Registro	Mpio	Dirección	Número de personas
1	1	—	3
2	2	—	2
3	1	—	5
4	3	—	1
5	4	—	6
6	4	—	8
7	4	—	4
8	1	—	3
9	2	—	5
10	4	—	7
11	3	—	3
12	3	—	1
13	1	—	6
14	1	—	2
15	2	—	8
16	4	—	9
17	1	—	4
18	3	—	10
19	2	—	7
20	1	—	2
21	EOF		

haga prueba de escritorio a los algoritmos del módulo:

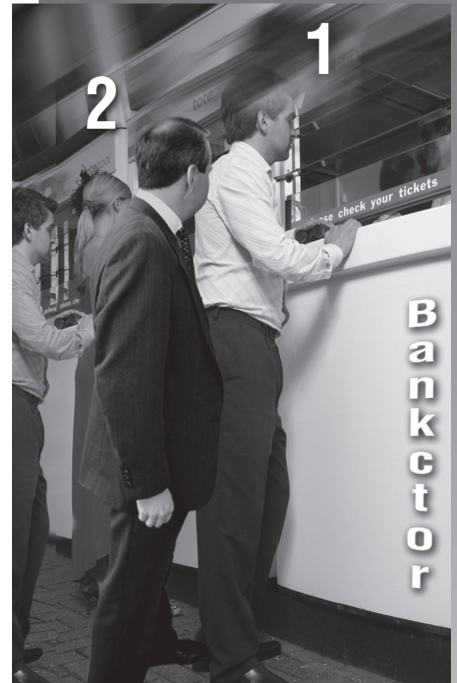
censoEnMedellin,
censoEnMedellinYBello
censoEnMedellinYSusVecinos

indicando claramente qué imprime en cada uno de ellos.

2. Explique los resultados obtenidos en las pruebas de escritorio efectuadas en el punto anterior.

Módulo 26

Definición y construcción



Introducción

En el módulo anterior introdujimos el concepto de arreglo de una dimensión, el cual se denomina vector. En este vamos a presentar una definición más formal acerca de lo que es un arreglo de una dimensión y las principales operaciones que podemos efectuar sobre él.

Objetivos del módulo

1. Definir, construir y manipular arreglos de una dimensión.

Preguntas básicas

1. ¿Qué es un vector?
2. ¿Cómo se recorre un vector?
3. ¿Con base en qué se define el tamaño de un vector?

Contenidos del módulo

26.1 Definición
26.2 Construcción del vector
26.3 Recorrido e impresión
26.4 Suma de los datos de un vector

«La forma como se construye una fila de personas se puede definir de varias maneras. Una de ellas es el orden cronológico. La primera persona que llega se ubica de primera, la segunda se ubica de segunda y así sucesivamente. Cada cliente que va llegando se ubica en el último lugar. Esta es una forma de construir este arreglo lineal de clientes».

Vea en el botón **Una dimensión** del mapa conceptual el video "Módulo 26. Concepto de arreglos: parte 2".

26.1 Definición

Un vector es una colección finita, homogénea y ordenada de datos del mismo tipo que se almacenan en posiciones consecutivas de memoria y que reciben un nombre común. Para referirse a un determinado elemento de un vector se debe utilizar un subíndice que especifique su posición en el arreglo.

26.2 Construcción del vector

Para trabajar con un vector, lo primero que se debe hacer es construir el vector. Para ilustrar dicha construcción consideremos el siguiente algoritmo, que llamaremos *censo*, el cual es una variación del algoritmo desarrollado en el módulo anterior:

```
1.    Algoritmo censo
2.          Variables: mpio, np, th, acmpio[125]: numericaEntera
3.                       dir: alfanumérica
4.          INICIO
5.                ABRA(censo)
6.                PARA  i  DESDE  1  HASTA  125  CON_VARIACION  +1
7.                      acmpio[i] = 0
8.                Fin(PARA)
9.                MIENTRAS  not  EOF(censo)
10.                     LEA(censo: mpio, dir, np)
11.                     acmpio[mpio] = acmpio[mpio] + np
12.               Fin(MIENTRAS)
13.               CIERRE(censo)
14.               imprime_vector(acmpio, 125)
15.               th = suma_vector(acmpio, 125)
16.         FIN
17.   Fin(censo)
```

En este nuevo algoritmo ya no utilizamos una variable para cada acumulador que se necesita manejar sino que definimos una sola variable, la cual llamamos **acmpio**, y que tiene características de vector, ya que tendrá 125 posiciones. Este tamaño (125 posiciones) se define con base en el conocimiento que se tiene acerca del problema que se va a resolver. En nuestro ejemplo debemos manejar el total de habitantes de cada uno de los 125 municipios que tiene el departamento y por lo tanto el vector de acumuladores deberá tener ese tamaño. Esta definición aparece en la instrucción 2 del algoritmo *censo*. Definimos la variable **acmpio** con una capacidad de 125 elementos.

En las instrucciones 6 a 8 inicializamos cada una de esas posiciones en cero.

En la instrucción 11 se configura cada una de las posiciones del vector **acmpio**, sumándole a la posición **mpio** el número de personas que se lee en cada registro.

En la instrucción 14 invocamos un subprograma llamado *imprimeVector*, con el cual recorremos e imprimimos los datos del vector que se ha construido.

En la instrucción 15 invocamos un subprograma llamado *sumaVector*, con el cual se suman todos los datos que se hallan en las diferentes posiciones del vector.

26.3 Recorrido e impresión

Ya hemos visto en nuestro algoritmo anterior que cuando se termina la construcción de un vector interesa conocer cuál es el valor almacenado en cada una de las posiciones del vector. Para efectuar esta tarea invocamos un subprograma denominado *imprime_vector*, el cual presentamos a continuación:

```
1.  void imprimeVector(V, n)
2.      Variables: i: numericaEntera
3.      INICIO
4.          PARA  i  DESDE  1  HASTA  n  CON_VARIACION  +1
5.              ESCRIBA(i, V[i])
6.          Fin(PARA)
7.      FIN
8.  Fin(imprimeVector)
```

Este subprograma es bastante simple: los parámetros son el nombre del vector y el tamaño del vector. Sólo se requiere una variable local para recorrer las diferentes posiciones del vector. A dicha variable la llamamos **i**.

En las instrucciones 4 a 6 escribimos un ciclo PARA, con el cual nos movemos en el vector desde la posición 1 hasta la posición **n**, imprimiendo el subíndice y el contenido de dicha posición.

26.4 Suma de los datos de un vector

Es también muy frecuente determinar el total de los datos almacenados en un vector. Para ello definimos un subprograma *función*, al cual llamamos *suma_vector*, que efectúe esta suma y retorne el resultado de ella al programa principal o llamante:

```
1.  entero sumaVector(V, n)
2.      Variables: i, s: numericaEntera
3.      INICIO
4.          s = 0
5.          PARA  i  DESDE 1 HASTA  n  CON_VARIACION  +1
6.              s = s + V[i]
7.          Fin(PARA)
8.          retorne(s)
9.      FIN
10. Fin(sumaVector)
```

Este subprograma es también bastante simple. Tiene como parámetros de entrada el nombre del vector y el número de elementos en él.

Se requieren dos variables locales: una para recorrer el vector y otra en la cual se maneje el acumulado. A dichas variables las llamamos **i** y **s**.

En la instrucción 4 se inicializa el acumulador **s** en cero.

En las instrucciones 5 a 8 escribimos nuestro ciclo de recorrido, y en vez de imprimir el contenido de cada posición, lo acumulamos en la variable llamada **s**.

En la instrucción 8 se retorna el contenido de **s** al programa llamante.

Resumen

En este módulo hemos presentado una definición más formal de lo que es un vector y la forma de construirlo para operar sobre él.

Ejercicios propuestos

1. Elabore un subprograma que imprima los datos que se hallan en las posiciones impares de un vector de **n** elementos.

2. Elabore un subprograma que imprima los datos que se hallan en las posiciones pares de un vector de **n** elementos.

3. Elabore un subprograma que calcule y retorne la suma de los datos múltiplos de 3 de un vector de **n** elementos.

4. Elabore un subprograma que calcule y retorne el promedio de los datos de un vector de **n** elementos.

5. Elabore un subprograma que calcule y retorne cuántas veces se halla un dato **d**, entrado como parámetro, en un vector de **n** elementos.

Módulo 27

Operaciones básicas: mayor dato, menor dato e intercambio de datos

Introducción

Cuando se trabaja con vectores existen ciertas necesidades que hay que suplir. Por ejemplo, si nuestro interés es ordenar los datos de un vector, intercambiar dos datos es una operación necesaria. Otras operaciones de gran uso son determinar el mayor dato y el menor dato de un vector. En este módulo trataremos los algoritmos correspondientes a estas necesidades.

Objetivos del módulo

1. Elaborar subprogramas con operaciones fundamentales en vectores.

Preguntas básicas

1. ¿Qué debe retornar un subprograma que determina el mayor dato de un vector?
2. ¿Por qué hay que intercambiar datos en un vector?

Contenidos del módulo

27.1 Determinar la posición en la cual se halla el mayor dato de un vector
27.2 Determinar la posición en la cual se halla el menor dato de un vector
27.3 Intercambiar dos datos en un vector

«Si la información que se maneja para cada uno de los clientes que se halla en una fila es el número de transacciones que va a realizar, nos podría interesar conocer en cuál posición se halla el cliente que tiene el menor número de transacciones, así como también nos puede interesar en cuál posición se halla el cliente que tiene el mayor número de transacciones. Según las políticas del banco se podrá tomar una decisión de a quién atender primero, lo cual implicará que las personas cambien de sitio en la fila».

Vea en el botón **Una dimensión** del mapa conceptual el video "Módulo 27. Manejo de vectores".

27.1 Determinar la posición en la cual se halla el mayor dato de un vector

Ya hemos visto que en un vector se almacenan los datos correspondientes a un conjunto de situaciones o eventos. Es importante conocer la situación o evento que más veces se presenta. Para lograr esto hay que elaborar un algoritmo que determine o identifique la posición en la cual se halla la situación que más veces ocurre. A continuación presentamos un algoritmo que efectúa dicha tarea:

```
1.   entero mayorDato(V, m)
2.        Variables: mayor, i: numericaEntera
3.        INICIO
4.             mayor = 1
5.             PARA i DESDE 2 HASTA m CON_VARIACION +1
6.                  SI V[i] > V[mayor]
7.                       mayor = i
8.                  Fin(SI)
9.             Fin(PARA)
10.            retorne(mayor)
11.       FIN
12.  Fin(mayorDato)
```

Para ilustrar cómo funciona este algoritmo consideremos la figura 27.1.

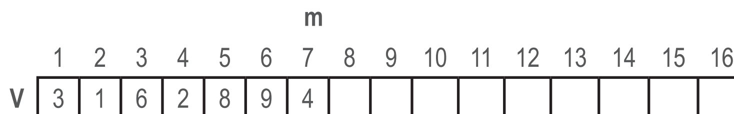


Figura 27.1

En la instrucción 2 se definen dos variables: **mayor,** para almacenar la posición en la cual se halla el dato mayor, e **i,** para recorrer el vector e ir comparando el dato de la posición **i** con el dato de la posición **mayor.**

En la instrucción 4 se inicializa la variable **mayor** en 1, definiendo que el dato mayor se halla en dicha posición.

En la instrucción 5 se plantea el ciclo con el cual se recorre el vector desde la posición 2 hasta la posición **m**.

En la instrucción 6 se compara el dato de la posición **i** con el dato de la posición **mayor**. Si el dato que está en la posición **i** es mayor que el dato que está en la posición **mayor** se actualiza el contenido de la variable **mayor** con la instrucción 7.

Al terminar el ciclo, cuando la **i** es mayor que la **m**, en la variable **mayor** estará almacenada la posición en la cual se halla el mayor dato del vector.

Para nuestro ejemplo, **mayor** se inicializa en 1, lo cual significa que el mayor dato está en la posición 1 del vector **V**, es decir, el mayor dato es **V[mayor]**, o sea 3.

Cuando se ejecuta el ciclo, **i** comienza valiendo 2, o sea que **V[i]** es 1. Al comparar **V[i]** con **V[mayor]** en la instrucción 6, el resultado es falso, por consiguiente no ejecuta la instrucción 7 y continúa con la instrucción 9, la cual incrementa el valor de **i** en 1 y regresa a la instrucción 5 a comparar **i** con **m**.

Como la **i** es menor o igual que la **m**, vuelve a ejecutar las instrucciones del ciclo: compara **V[i]** con **V[mayor]**, es decir, compara **V[3]** con **V[1]**. El dato de la posición **i** es mayor que el dato de la posición **mayor**, por lo tanto ejecuta la instrucción 7, o sea que modifica el contenido de la variable **mayor**, el cual será 3, indicando que el mayor dato se halla en la posición 3 del vector.

Se llega de nuevo a la instrucción 9 e incrementa la **i** en 1 (**i** queda valiendo 4) y regresa de nuevo a la instrucción 5 a continuar ejecutando las instrucciones del ciclo hasta que la **i** sea mayor que la **m**.

Cuando la **i** sea mayor que la **m** termina la ejecución del ciclo y en la variable **mayor** quedará la posición en la cual se halla el mayor dato del vector.

Observe que hemos trabajado con las posiciones del vector, ya que con ellas tenemos acceso directo a los datos que se hallan en dichas posiciones.

27.2 Determinar la posición en la cual se halla el menor dato de un vector

En el numeral anterior desarrollamos un algoritmo en el cual se determina la posición en la cual se halla el mayor dato de un vector. Es también necesario, en muchas situaciones, determinar la posición en la cual se halla el menor dato. A continuación se presenta un algoritmo que efectúa esta tarea:

```
1.   entero menorDato(V, m)
2.       Variables: menor, i: numericaEntera
3.       INICIO
4.           menor = 1
5.           PARA i DESDE 2 HASTA m CON_VARIACION +1
6.               SI V[i] < V[menor]
7.                   menor = i
8.               Fin(SI)
9.           Fin(PARA)
10.          retorne(menor)
11.      FIN
12.  Fin(menorDato)
```

Este algoritmo es similar al algoritmo de determinar la posición en la cual se halla el mayor dato. La única diferencia entre este y el anterior es que la variable en la cual se retorna el resultado de la búsqueda se denomina **menor**, en vez de **mayor**, y la comparación de **V[i]** con **V[menor]** se hace con el símbolo menor que (<), en vez del símbolo mayor que (>). Dejamos al estudiante, como ejercicio, comprobar el correcto funcionamiento de dicho algoritmo.

27.3 Intercambiar dos datos en un vector

Intercambiar dos datos en un vector es una operación frecuente en manipulación de vectores. Por ejemplo, cuando se desean ordenar los datos de un vector es necesario cambiar los datos de posición, y este cambio se hace por parejas. Un algoritmo que efectúa dicha tarea es el siguiente:

```
1.  void intercambio(V, i, j)      // V: parámetro por referencia
2.      Variables: aux: numericaEntera
3.      INICIO
4.          aux = V[i]
5.          V[i] = V[j]
6.          V[j] = aux
7.      FIN
8.  Fin(intercambio)
```

En la instrucción 1 se define el subprograma con sus parámetros: **V**, el nombre del vector en el cual se desea hacer el intercambio, e **i** y **j**, las posiciones cuyos datos se desean intercambiar.

En la instrucción 2 se define una variable, llamada **aux**, que se utilizará para efectuar el intercambio.

En la instrucción 4 se guarda el contenido de la posición **i** del vector en la variable auxiliar **aux** con el fin de que dicho dato no se pierda.

En la instrucción 5 se lleva a la posición **i** del vector lo que hay en la posición **j** del vector.

En la instrucción 6 se lleva lo que hay en la variable auxiliar, es decir, lo que había en la posición **i** del vector, a la posición **j** del vector.

Como ejemplo, consideremos el vector de la figura 27.2, y que se desea intercambiar el dato que se halla en la posición 2 (**i == 2**) con el dato que se halla en la posición 5 (**j == 5**).

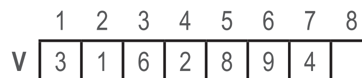


Figura 27.2

Al ejecutar nuestro subprograma *intercambio*, el vector quedará como en la figura 27.3.

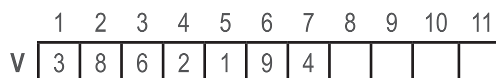


Figura 27.3

Resumen

En este módulo hemos presentado algoritmos de algunas de las operaciones básicas en vectores: determinar la posición en la cual se halla el mayor dato del vector, determinar la posición en la cual se halla el menor dato del vector e intercambiar el contenido de dos celdas del vector.

Ejercicios propuestos

1. Elabore un subprograma que intercambie los datos de un vector así: el primero con el segundo, el tercero con el cuarto, el quinto con el sexto, y así sucesivamente.

2. Elabore un subprograma que intercambie los datos de un vector así: el primero con el último, el segundo con el penúltimo, el tercero con el antepenúltimo, y así sucesivamente.

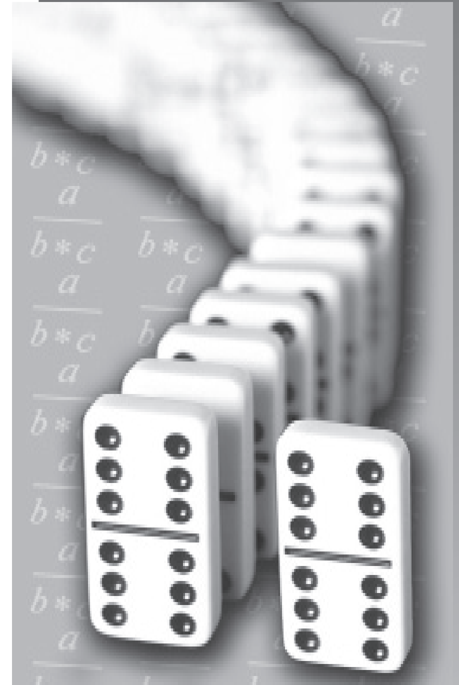
3. Elabore un subprograma que intercambie los datos de un vector así: el primero con el tercero, el segundo con el cuarto, el quinto con el séptimo, el sexto con el octavo, y así sucesivamente.

4. Elabore subprogramas para los tres ejercicios anteriores usando sólo una variable para el recorrido.

5. Elabore un algoritmo que intercambie el mayor dato con el menor en un vector de **n** elementos.

Módulo 28

Operaciones básicas: proceso de inserción en un vector ordenado ascendentemente



Introducción

Cuando se maneja un gran volumen de datos, una de las operaciones más frecuentes es buscar un dato. Este proceso de búsqueda será eficiente si los datos están ordenados bajo algún criterio. Imagínese el proceso de búsqueda de un teléfono en un directorio telefónico en el cual los nombres de las personas no estén ordenados.

Cuando se tiene un conjunto de datos ordenado, una operación frecuente es añadir un nuevo dato a este conjunto de datos. Para conservar la cualidad de que la búsqueda siga siendo eficiente, debemos añadir este nuevo dato ubicándolo en un sitio tal que los datos conserven el orden establecido.

Los pasos para añadir un nuevo dato son buscar dónde insertar e insertar. En este módulo trataremos los algoritmos para efectuar este proceso.

«Si la política de un banco es atender a los clientes en orden ascendente según el número de transacciones que vayan a efectuar, cuando llegue un nuevo cliente habrá que ubicarlo en la fila en el sitio correcto, de acuerdo al número de transacciones que tenga que hacer. Para ello debemos buscar en cuál sitio debe quedar y ubicarlo entonces en ese sitio. La anterior operación implica que los clientes que están en la fila y que tengan más transacciones que el nuevo cliente van a quedar en una posición más atrás en la fila».

Objetivos del módulo

1. Elaborar subprogramas para el proceso de inserción de un dato en un vector ordenado ascendentemente.

Preguntas básicas

1. ¿Por qué manejar los datos ordenados?
2. ¿Cómo se conservan ordenados los datos al insertar un nuevo dato?
3. ¿Qué hay que hacer para que los datos sigan ordenados al insertar un nuevo dato?

Contenidos del módulo

28.1 Buscar dónde insertar un dato en un vector ordenado ascendentemente
28.2 Insertar un dato en un vector ordenado ascendentemente

Vea en el botón **Una dimensión** del mapa conceptual el video "Módulo 28. Proceso de inserción".

28.1 Buscar dónde insertar un dato en un vector ordenado ascendentemente

Para buscar dónde insertar un nuevo dato en un vector ordenado ascendentemente debemos recorrer el vector e ir comparando el dato de cada posición con el dato a insertar. Como los datos están ordenados ascendentemente, cuando se encuentre en el vector un dato mayor que el que se va a insertar esa es la posición en la cual deberá quedar el nuevo dato. Si el dato a insertar es mayor que todos los datos del vector, entonces el dato a insertar quedará de último.

El algoritmo siguiente ejecuta esta tarea y retorna en la variable **i** la posición en la cual debe quedar el dato a insertar:

```
1.    Entero buscarDondeInsertar(V, m, d)      // V, m, d: parámetros por valor
2.          Variables: i: numericaEntera
3.          INICIO
4.                i = 1
5.                MIENTRAS (i <= m && V[i] < d)
6.                      i = i + 1
7.                Fin(MIENTRAS)
8.                retorne(i)
9.          FIN
10.   Fin(buscarDondeInsertar)
```

En la instrucción 1 se define el nombre del subprograma con sus parámetros: **V**, variable que contiene el nombre del vector en el cual hay que efectuar el proceso de búsqueda; **m**, variable que contiene el número de posiciones utilizadas en el vector; y **d**, variable que contiene el dato a insertar.

En la instrucción 2 se define la variable **i**, que es la variable con la que se recorre el vector y se va comparando el dato de la posición **i** del vector (**V[i]**) con el dato a insertar **d**.

En la instrucción 4 se inicializa la variable **i** en 1, ya que esta es la posición a partir de la cual se comienzan a almacenar los datos.

En la instrucción 5, la instrucción del ciclo, se controlan dos situaciones: una, que no se haya terminado de comparar todos los datos del vector, (**i <= m**), y dos, que el dato de la posición **i** sea menor que **d** (**V[i] < d**).

Si ambas condiciones son verdaderas se ejecutan las instrucciones 6 y 7, es decir, se incrementa la **i** en 1 y se regresa a la instrucción 5 a evaluar nuevamente las condiciones.

Cuando una de las condiciones sea falsa (**i > m** o **V[i] >= d**), se sale del ciclo y ejecuta la instrucción 8, es decir, retorna al programa llamante el valor de **i**: la posición en la cual hay que insertar el dato **d**.

Como ejemplo, consideremos el vector de la figura 28.1, y que se desea insertar el número 10 (**d == 10**).

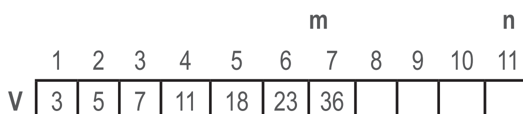


Figura 28.1

Al ejecutar el algoritmo *buscarDondeInsertar*, éste retorna **i** valiendo 4, es decir, en la posición 4 del vector **V** debe quedar el dato 10.

En la figura 28.2 se muestra esta situación:

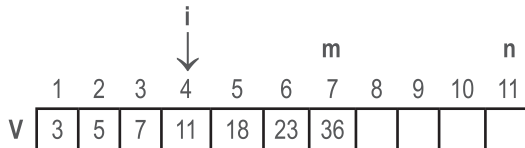


Figura 28.2

28.2 Insertar un dato en un vector ordenado ascendentemente

Conociendo que el dato a insertar debe quedar en una posición **i**, veamos cómo se efectúa este proceso. Si el vector no está lleno (**m == n**), debemos mover todos los datos desde la posición **i** hasta la posición **m** una posición hacia la derecha y luego almacenar en la posición **i** del vector el dato **d**. Esta tarea la efectuamos con el siguiente algoritmo:

```
1.   void insertar(V, m, n, i, d)        // V, m: parámetros por referencia
2.       Variables: j: numericaEntera
3.       INICIO
4.           SI m == n
5.               ESCRIBA("vector lleno")
6.               retorne
7.           Fin(SI)
8.           PARA j DESDE m HASTA i CON_VARIACION –1
9.               V[j + 1] = V[j]
10           Fin(PARA)
11.          V[i] = d
12.          m = m + 1
13.      FIN
14.  Fin(insertar)
```

En la instrucción 1 se define el subprograma con sus parámetros: **V**, variable que contiene el nombre del vector en el cual hay que efectuar el proceso de inserción; **m**, variable que contiene el número de posiciones utilizadas en el vector; **n**, tamaño del vector; **i**, variable que contiene la posición en la cual debe quedar el dato contenido en **d**; y **d**, variable que contiene el dato a insertar. Es bueno resaltar que los parámetros **V** y **m** deben ser parámetros por referencia, mientras que **n**, **i** y **d** son por valor.

En la instrucción 2 se define la variable **j**, la cual se utilizará para mover los datos del vector en caso de que sea necesario.

En la instrucción 4 se controla que el vector no esté lleno. En caso de que el vector esté lleno (**m == n**), se ejecutan las instrucciones 5 y 6, las cuales producen el mensaje de que el vector está lleno y retornan el control al programa llamante.

En caso de que el vector no esté lleno se ejecuta la instrucción 8.

En la instrucción 8 se plantea la instrucción PARA, la cual inicializa el valor de **j** con el contenido de **m,** compara el valor de **j** con el valor de **m** y define la forma

como variará el valor de **j**. La variación de la variable **j** será restándole 1 cada vez que llegue a la instrucción fin del PARA. Cuando el valor de **j** sea menor que **i** se sale del ciclo. En caso de que el valor inicial de **j** sea menor que **i**, no ejecutará la instrucción 9 y continuará con la instrucción 11.

En la instrucción 9 se mueve el dato que está en la posición **j** hacia la posición **j + 1**.

En la instrucción 10, que es el fin de la instrucción PARA, disminuye el valor de **j** en 1 y regresa a la instrucción 8 a comparar el valor de **j** con **i**. Si la **i** es mayor o igual que la **j**, ejecutará nuevamente las instrucciones 9 y 10, hasta que la **i** sea menor que la **j**.

Cuando se sale del ciclo, ejecuta la instrucción 11, en la cual se asigna a la posición **i** del vector el dato **d** y continúa con la instrucción 12 en la cual se incrementa el valor de **m** en 1, indicando que el vector ha quedado con un elemento más.

Para nuestro ejemplo, al terminar de ejecutar las instrucciones del ciclo el vector está como en la figura 28.3, y al ejecutar las instrucciones 11 y 12 el vector queda como en la figura 28.4.

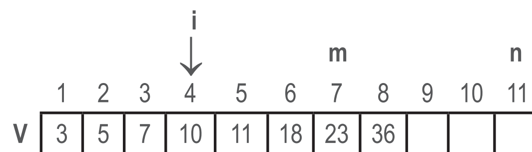


Figura 28.3

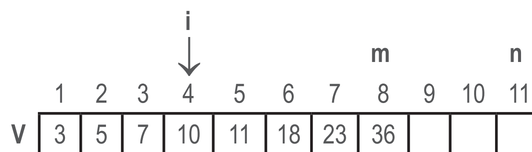


Figura 28.4

Resumen

En este módulo hemos presentado los algoritmos correspondientes al proceso de insertar un nuevo dato en un vector que tiene los datos ordenados en forma ascendente. Dicho proceso de inserción comprende los algoritmos de buscar dónde insertar e insertar.

Ejercicios propuestos

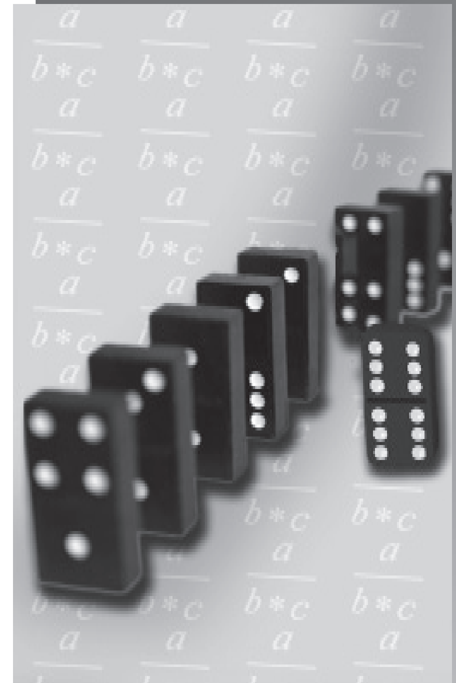
1. Considerando los subprogramas para el proceso de inserción, vistos en el módulo, si se desea insertar el dato **d == 4** en el vector de la figura 28.4, ¿en cuál posición queda? Explique su respuesta.

2. Si el vector en el cual se va a insertar un nuevo dato está vacío, ¿cuál es valor de **i** que retorna el subprograma *buscarDondeInsertar*?

3. Si el dato a insertar es mayor que todos los datos que hay en el vector, ¿cuánto retorna valiendo **i** el subprograma *buscarDondeInsertar*?

Módulo 29

Operaciones básicas: proceso de borrado en un vector



Introducción

Así como el proceso de inserción es muy frecuente cuando se manejan grandes volúmenes de datos, el proceso de borrado también lo es. Este proceso implica dos tareas: buscar el dato a borrar y borrarlo. En este módulo nos ocuparemos de los algoritmos que ejecuten estas tareas.

Objetivos del módulo

1. Elaborar subprogramas para el proceso de borrado de un dato en un vector.

Preguntas básicas

1. ¿Cómo se identifica que el dato a borrar no se encuentra en el vector?
2. ¿Hay que mover datos en el vector cuando se borra un dato?

Contenidos del módulo

29.1 Buscar dato a borrar
29.2 Borrar dato

«Consideremos de nuevo el ejemplo del módulo anterior, en el que la norma de un banco es atender a los clientes en orden ascendente según el número de transacciones que vayan a efectuar. Puede suceder entonces que un cliente se aburra en la fila y que desista de hacer sus transacciones en ese momento. Dicho cliente abandona la fila. Esta decisión implica que los clientes que estaban ubicados detrás del cliente que abandonó la fila van a quedar en una posición más cerca de la taquilla».

Vea en el botón **Una dimensión** del mapa conceptual el video "Módulo 29. Proceso de borrado".

29.1 Buscar dato a borrar

El proceso para determinar en cuál posición de un vector se halla un dato es bastante similar al proceso de buscar en dónde insertar. En el siguiente algoritmo presentamos dicho proceso:

```
1.   entero buscarPosicionDato(V, m, d)
2.        Variables: i: numericaEntera
3.        INICIO
4.             i = 1
5.             MIENTRAS (i <= m && V[i] ≠ d)
6.                  i = i + 1
7.             Fin(MIENTRAS)
8.             retorne(i)
9.        FIN
10.  Fin(buscarPosicionDato)
```

La única diferencia de este algoritmo con el algoritmo para buscar dónde insertar es que en la instrucción MIENTRAS la comparación de **V[i]** con **d** se efectúa con el operador diferente (≠) y no con el operador menor que (**<**).

Nuestro algoritmo de buscar dato funciona independientemente de que los datos se encuentren ordenados o no. En caso de que el dato a buscar no se halle en el vector, nuestro algoritmo retorna en la variable **i** el valor de **m + 1**, lo cual usaremos como condición para detectar si el dato que se está buscando se halla en el vector o no. En otras palabras, si **i** es igual a **m + 1** el dato no está en el vector **V**.

Como ejemplo, consideremos el vector de la figura 29.1, y que deseamos borrar el dato 2:

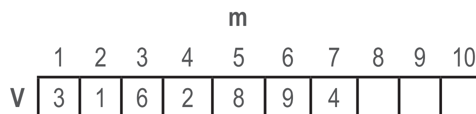


Figura 29.1

Al ejecutar nuestro algoritmo *buscarPosicionDato*, el contenido de la variable **i** será 4, es decir, en la posición 4 del vector se halla el dato 2. En la figura 29.2 presentamos esta situación:

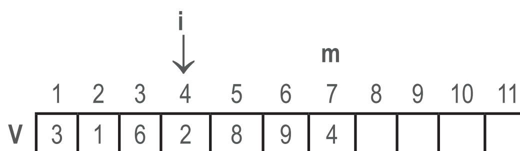


Figura 29.2

29.2 Borrar dato

Conociendo la posición **i** en la cual se halla un dato a borrar, el proceso de borrado implica que hay que mover todos los datos que se encuentran desde la posición **i + 1** hasta la posición **m** una posición hacia la izquierda y restarle 1 a **m**. En el siguiente algoritmo se efectúa dicho proceso:

```
1.  void borrar(V, m, i)  // V, m: parámetros por referencia
2.        Variables: j: numericaEntera
3.        INICIO
4.              SI i == m + 1
5.                    ESCRIBA("dato no existe")
6.                    retorne
7.              Fin(SI)
8.              PARA j DESDE i + 1 HASTA m CON_VARIACION +1
9.                    V[j – 1] = V[j]
10.             Fin(PARA)
11.             m = m – 1
12.       FIN
13. Fin(borrar)
```

En la instrucción 1 se define el subprograma con sus parámetros: **V**, variable que contiene el nombre del vector en el cual hay que efectuar el proceso de borrar; **m**, variable que contiene el número de posiciones utilizadas en el vector; e **i**, variable que contiene la posición del dato a borrar.

En la instrucción 2 se define la variable **j**, la cual se utilizará para mover los datos del vector en caso de que sea necesario.

En la instrucción 4 se controla que el dato a borrar exista en el vector. En caso de que el dato no exista (**i == m +1**) se ejecutan las instrucciones 5 y 6, las cuales producen el mensaje de que el dato no está en el vector y retornan el control al programa llamante.

En caso de que el dato esté en el vector se ejecuta la instrucción 8.

En la instrucción 8 se plantea la instrucción PARA, la cual inicializa el valor de **j** con el contenido de **i + 1,** compara el valor de **j** con el valor de **m** y define la forma como variará el valor de **j**. La variación de la variable **j** será sumándole 1 cada vez que llegue a la instrucción FIN del PARA. Cuando el valor de **j** sea mayor que **m** se sale del ciclo. En caso de que el valor inicial de **j** sea menor que **m** no ejecutará la instrucción 9 y continuará con la instrucción 11.

En la instrucción 9 se mueve el dato que está en la posición **j** hacia la posición **j – 1**.

En la instrucción 10, que es el fin de la instrucción PARA, incrementa el valor de **j** en 1 y regresa a la instrucción 8 a comparar el valor de **j** con **m**. Si **j** es mayor que **m**, ejecutará nuevamente las instrucciones 9 y 10, hasta que **j** sea mayor que **m**.

Cuando se sale del ciclo, ejecuta la instrucción 11, en la cual se le resta 1 al valor de **m** indicando que el vector ha quedado con un elemento menos.

Para nuestro ejemplo, al terminar de ejecutar el subprograma *borrar* el vector queda como en la figura 29.3:

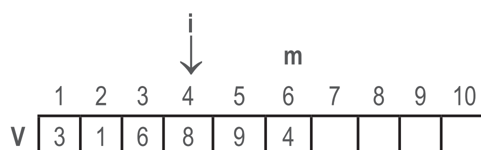


Figura 29.3

Es conveniente, en este punto, hacer notar algo que es muy importante: en la figura 29.3, y sólo con fines explicativos, hemos dejado el contenido de la posición 7 en blanco, indicando que se ha borrado un dato. En realidad, en la posición 7 del vector seguirá estando un 4 hasta que se reemplace por otro valor, pero el contenido de **m** se ha disminuido en 1, lo cual significa que el dato de la posición 7 ya no pertenece al vector.

Resumen

En este módulo hemos presentado los algoritmos correspondientes al proceso de borrar un dato en un vector. Dicho proceso de borrado comprende los algoritmos de buscar la posición en la cual se halla el dato y eliminarlo del vector.

Ejercicios propuestos

1. Elabore un subprograma que elimine de un vector todas las ocurrencias de un dato **d** entrado como parámetro.

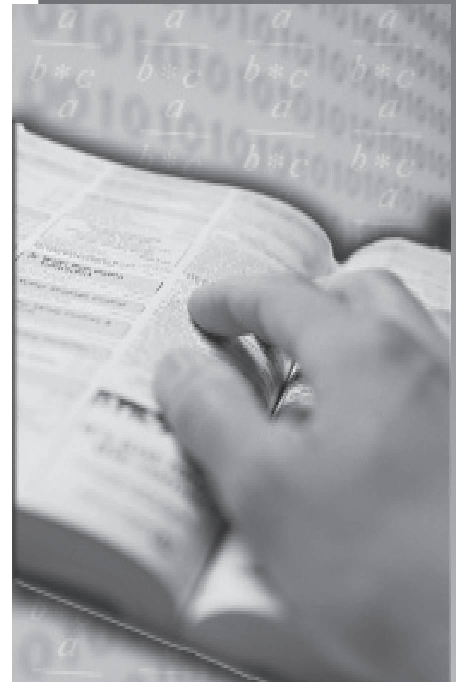
2. Se tiene un vector cuyos datos se hallan ordenados en forma descendente. Elabore un subprograma que procese dicho vector, de tal manera que no quede con datos repetidos, es decir, si hay algún dato repetido, sólo debe quedar uno de ellos.

3. Elabore un algoritmo que ejecute la misma tarea del ejercicio anterior considerando que los datos no se hallan ordenados bajo ningún criterio.

4. Elabore un subprograma que sustituya un valor **x** por un valor **y** en un vector cuyos datos se hallan ordenados ascendentemente, de tal manera que el vector siga conservando su orden creciente. La función debe regresar como resultado 1 si se hizo la sustitución, 0 si **x** no se encontraba en el arreglo y –1 si el arreglo estaba vacío.

Módulo 30

Operaciones básicas: búsqueda binaria y ordenamiento por selección



Introducción

En el módulo anterior se presentó un algoritmo para buscar un dato en un vector. En este algoritmo la búsqueda se efectuaba desde el primer dato y se iba comparando el dato de cada posición del vector con el dato que se deseaba hallar. Los datos en el vector, se dijo, no estaban ordenados bajo ningún criterio; por consiguiente, para detectar que el dato que se está buscando no se encuentre en el vector hay que comparar el dato que se está buscando con todos los datos del vector. Si el vector tuviera un millón de datos se deben hacer un millón de comparaciones para poder decir que el dato no está en el vector. Si los datos estuvieran ordenados el proceso de búsqueda sería distinto y la eficiencia sería mejor.

En este módulo presentamos un método eficiente de búsqueda de un dato en un vector, considerando que los datos del vector se hallan ordenados en forma ascendente.

«En la vida diaria una de las actividades que es necesario efectuar muchas veces es buscar algún dato. Por ejemplo, si deseamos conocer el teléfono de una persona acudimos al directorio telefónico y aprovechamos el hecho de que sus nombres están ordenados alfabéticamente. Esta característica facilita el proceso de búsqueda. Si los nombres no estuvieran ordenados el proceso de buscar el teléfono de una persona sería una tarea supremamente dispendiosa. La búsqueda binaria es un método que aprovecha la característica de que los datos se encuentren ordenados».

Objetivos del módulo

1. Elaborar un algoritmo eficiente para el proceso de búsqueda en un vector ordenado.
2. Ordenar los datos de un vector aplicando el método de selección.

Preguntas básicas

1. ¿Por qué la búsqueda binaria es eficiente?
2. ¿En qué consiste la búsqueda binaria?
3. ¿Cómo funciona el método de ordenamiento por selección?

Contenidos del módulo

30.1 Búsqueda binaria
30.2 Ordenamiento por selección

Vea en el botón **Una dimensión** del mapa conceptual el video "Módulo 30. Ordenamiento por selección".

30.1 Búsqueda binaria

El proceso de búsqueda binaria utiliza el hecho de que los datos se encuentran ordenados en forma ascendente. La primera comparación se hace con el dato de la mitad del vector. Si se tiene suerte, ése es el que se está buscando y no hay que hacer más comparaciones; si no lo es, es porque el que se está buscando es mayor o menor que el dato de la mitad del vector. Si el que se está buscando es mayor que el de la mitad del vector, significa que si el dato se halla en el vector, está a la derecha del de la mitad, o sea que no habrá necesidad de comparar el dato que se está buscando con todos los datos que están a la izquierda del dato de la mitad del vector. Luego de que hemos descartado la mitad de los datos, la siguiente comparación se hará con el de la mitad, de la mitad en la cual posiblemente esté el dato a buscar. Así continuamos con este proceso hasta que se encuentre el dato, o se detecte que no está en el vector. Si tuviéramos un millón de datos, con una sola comparación estamos ahorrando 500.000 comparaciones, con la segunda se ahorran 250.000 comparaciones, con la tercera se ahorran 125.000 y así sucesivamente. Como se podrá ver la reducción del número de comparaciones es notable. A continuación se presenta un algoritmo con el cual se efectúa este proceso:

```
1.   entero busquedaBinaria(V, n, d)
2.       Variables: i, j, m: numericaEntera
3.       INICIO
4.           i = 1
5.           j = n
6.           MIENTRAS  i <= j
7.               m = (i + j) / 2
8.               SI V[m] == d
9.                   retorne(m)
10.              Fin(SI)
11.              SI V[m] < d
12.                  i = m + 1
13.              DE_LO_CONTRARIO
14.                  j = m – 1
15.              Fin(SI)
16.          Fin(MIENTRAS)
17.          retorne(n + 1)
18.      FIN
19.  Fin(busquedaBinaria)
```

En la instrucción 1 se define el subprograma *busquedaBinaria(V, n, d)*, cuyos parámetros son: **V**, el vector en el cual hay que efectuar la búsqueda; **n**, el tamaño del vector; y **d**, el dato a buscar.

En la instrucción 2 se definen las variables de trabajo: **i**, para guardar la posición inicial del rango en el cual hay que efectuar la búsqueda; **j**, para guardar la posición final del rango en el cual hay que efectuar la búsqueda; y **m**, para guardar la posición de la mitad del rango entre **i** y **j**.

En las instrucciones 4 y 5 se asignan los valores iniciales a las variables **i** y **j**. Estos valores iniciales son 1 y **n**, ya que la primera vez el rango sobre el cual hay que efectuar la búsqueda es desde la primera posición hasta la última del vector.

En la instrucción 6 se plantea el ciclo MIENTRAS, con la condición de que **i** sea menor o igual que **j**. Es decir, si el límite inferior (**i**) es menor o igual que el límite superior (**j**), aún existen posibilidades de que el dato que se está buscando se encuentre en el vector. Cuando **i** sea mayor que **j** significa que el dato no está en el vector y retornará **n + 1**.

Cuando la condición de la instrucción 6 sea verdadera se ejecutan las instrucciones del ciclo.

En la instrucción 7 se calcula la posición de la mitad entre **i** y **j**. Llamamos **m** a esta posición.

En la instrucción 8 se compara el dato de la posición **m** con el dato **d**. Si el dato de la posición **m** es igual al dato **d** que se está buscando (**V[m] == d**), ejecuta la instrucción 9: termina el proceso de búsqueda retornando **m**, la posición en la cual se halla el dato **d** en el vector.

En caso de que el dato de la posición **m** sea diferente de **d**, se ejecuta la instrucción 11, en la cual se compara si el dato de la posición **m** es menor que el dato **d**.

Si el resultado de esta comparación es verdadero, significa que si el dato está en el vector, se halla a la derecha de la posición **m**; por consiguiente, el valor inicial del rango en el cual se debe efectuar la búsqueda es **m + 1**, y por lo tanto ejecuta la instrucción 12 en la cual a la variable **i** se le asigna **m + 1**.

Si el resultado de la comparación de la instrucción 11 es falso, significa que si el dato está en el vector, se halla a la izquierda de la posición **m**; por consiguiente, el valor final del rango en el cual se debe efectuar la búsqueda es **m – 1**, y por lo tanto ejecuta la instrucción 14 en la cual a la variable **j** se le asigna **m – 1**.

Habiendo actualizado el límite inicial o el límite final, se llega a la instrucción 16, la cual retorna la ejecución a la instrucción 6, donde se efectúa de nuevo la comparación entre **i** y **j**. Cuando la condición sea falsa se sale del ciclo y ejecuta la instrucción 17, la cual retorna **n + 1**, indicando que el dato no se halla en el vector.

Fíjese que la única manera de que se ejecute la instrucción 17 es que no haya encontrado el dato que se está buscando, ya que si lo encuentra ejecuta la instrucción 9, la cual termina el proceso.

Consideremos, como ejemplo, el vector de la figura 30.1, y que se desea buscar el número 38 en dicho vector:

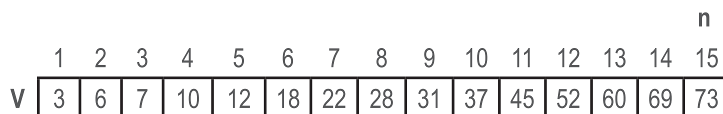


Figura 30.1

Al ejecutar nuestro algoritmo de búsqueda binaria, y cuando se esté en la instrucción 7 por primera vez, la situación en el vector se muestra en la figura 30.2:

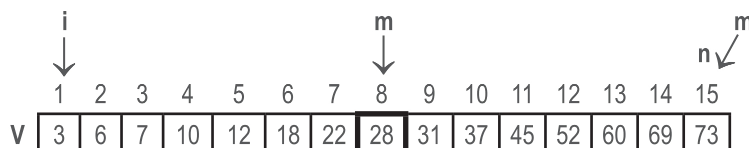


Figura 30.2

Al ejecutar la instrucción 8, la cual compara el dato de la posición **m** (**V[m] == 28**) con **d** (**d == 38**), se determina que el dato **d**, si está en el vector, se halla a la derecha de la posición **m**, por consiguiente el valor de **i** deberá ser 9. En consecuencia, al ejecutar las instrucciones 11 y 12, el límite inicial del rango sobre el cual hay que efectuar la búsqueda es 9, y el rango sobre el cual hay que efectuarla es entre 9 y 15. La situación en el vector es como en la figura 30.3:

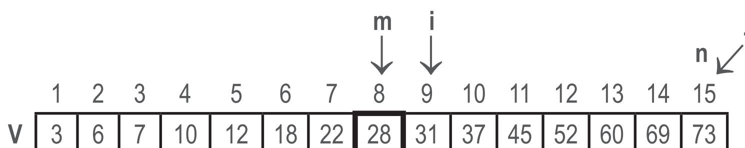


Figura 30.3

Al ejecutar el ciclo por segunda vez el valor de **m** será 12, y la situación en el vector se muestra en la figura 30.4:

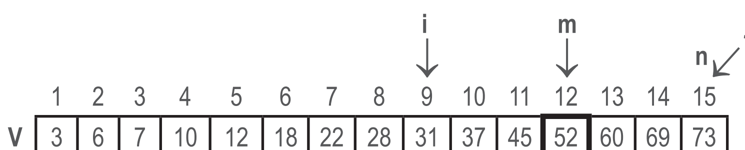


Figura 30.4

Luego, al comparar **d** (**d == 38**) con **V[m]** (**V[m] == 52**), se determina que el dato que se está buscando, si está en el vector, debe hallarse a la izquierda de **m**, es decir, entre las posiciones 9 y 11, por consiguiente el valor de **j** será 11 (**j == m – 1**).

Al ejecutar el ciclo por tercera vez, la situación del vector es como en la figura 30.5:

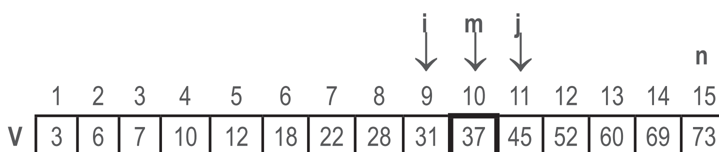


Figura 30.5

En esta pasada se detecta que el dato debe estar a la derecha de **m**, por consiguiente el valor de **i** deberá ser 11, y en consecuencia el valor de **m** también. La situación en el vector es como en la figura 30.6:

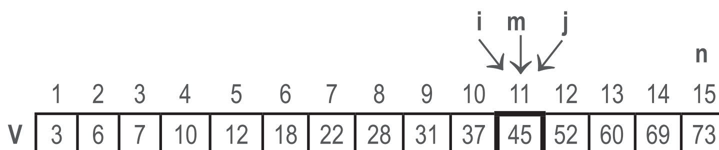


Figura 30.6

En este punto la comparación de **d** con **V[m]** indica que si el dato **d** está en el vector, debe estar a la izquierda de **m**, por lo tanto el valor de **j** será 10, y al regresar a la instrucción 6 y evaluar la condición del ciclo (**i <= j**), ésta es falsa, por consiguiente termina el ciclo y retorna en **m** el valor de 16 (**n + 1**), indicando que el 38 no se encuentra en dicho vector.

30.2 Ordenamiento por selección

Como ya hemos visto, el hecho de trabajar los datos ordenados hace que los algoritmos sean más eficientes. Para ejecutar el proceso de ordenamiento se han desarrollado múltiples algoritmos, buscando que este proceso sea lo más eficiente posible. Vamos a presentar en este módulo un algoritmo que ordena los datos de un vector. El método utilizado en este algoritmo se denomina ordenamiento por selección.

Este método se enuncia así: de los datos que faltan por ordenar determinar el menor de ellos y ubicarlo de primero en ese conjunto de datos. Con base en este enunciado hemos escrito el algoritmo que se presenta a continuación:

```
1.   void ordenamientoAscendenteSeleccion(V, m)
2.       Variables: i, j, k: numericaEntera
3.       INICIO
4.           PARA  i  DESDE 1 HASTA m – 1  CON_VARIACION  +1
5.               k = i
6.               PARA  j  DESDE  i + 1 HASTA  m  CON_VARIACION  +1
7.                   SI  V[j] < V[k]
8.                       k = j
9.                   Fin(SI)
10.              Fin(PARA)
11.              intercambiar(V, k, i)
12.          Fin(PARA)
13.      FIN
14.  Fin(ordenamientoAscendenteSeleccion)
```

En la instrucción 1 se define el subprograma con sus parámetros: **V**, variable en la cual se almacena el vector que se desea ordenar, y **m**, el número de elementos a ordenar. **V** es parámetro por referencia.

En la instrucción 2 se definen las variables necesarias para efectuar el proceso de ordenamiento. La variable **i** se utiliza para identificar a partir de cuál posición es que faltan datos por ordenar. Inicialmente el valor de **i** es 1, ya que inicialmente faltan todos los datos por ordenar y los datos comienzan en la posición 1. Cuando el contenido de la variable **i** sea 2, significa que faltan por ordenar los datos que hay a partir de la posición 2 del vector; cuando el contenido de la variable **i** sea 4, significa que faltan por ordenar los datos que hay a partir de la posición 4; cuando el contenido de la variable **i** sea **m**, significa que estamos en el último elemento del vector, el cual obviamente estará en su sitio, pues no hay más datos con los cuales se pueda comparar. Esta es la razón por la cual en la instrucción 4 se pone a variar la **i** desde 1 hasta **m – 1**.

En la instrucción 4 se plantea el ciclo de la variable **i**, que variará desde 1 hasta **m – 1**, tal como explicamos en el párrafo anterior.

En la instrucción 5 se le asigna a **k** el contenido de la variable **i**. La variable **k** se utiliza para identificar la posición en la que se halla el menor dato. Inicialmente suponemos que el menor dato se encuentra en la posición **i** del vector, es decir, suponemos que es el primero del conjunto de datos que faltan por ordenar.

En la instrucción 6 se plantea el ciclo con el cual se determina la posición en la cual se halla el menor dato del conjunto de datos que faltan por ordenar. En este ciclo se utiliza la variable **j**, que tiene un valor inicial de **i + 1** y varía hasta **m**.

En la instrucción 7 se compara el dato que se halla en la posición **j** del vector con el dato que se halla en la posición **k**. Si el contenido de la posición **j** es menor que el contenido de la posición **k**, se reemplaza el contenido de la variable **k** por **j**. De esta manera en la variable **k** siempre estará la posición en la cual se encuentra el menor dato.

Al terminar la ejecución del ciclo interno, en la variable **k** estará la posición en la cual se halla el menor dato.

En la instrucción 11 se intercambia el dato que se halla en la posición **k** con el que se halla en la posición **i**, logrando de esta manera ubicar el menor dato al principio del conjunto de datos que faltan por ordenar.

Al llegar a la instrucción 12 continúa con el ciclo externo incrementando **i** en 1, indicando que es a partir de esa posición que faltan datos por ordenar.

Como ejemplo, consideremos el vector de la figura 30.7:

Figura 30.7

Al ejecutar por primera vez las instrucciones 4, 5 y 6 los valores de **i**, **k** y **j** son 1, 1 y 2, respectivamente, lo cual indica: faltan por ordenar los datos a partir de la posición 1; el menor dato, de los datos que faltan por ordenar, se halla en la posición 1 del vector; se va a comenzar a comparar los datos del vector, a partir de la posición 2. Gráficamente se presenta esta situación en la figura 30.8:

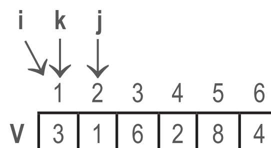


Figura 30.8

Al ejecutar la instrucción 7, por primera vez, compara el dato de la posición 2 **(j == 2)** con el dato de la posición 1 **(k == 1)**, obteniendo como resultado que la condición es verdadera; por lo tanto, ejecuta la instrucción 8, la cual asigna a **k** el contenido de la variable **j**, es decir, 2. Por consiguiente el contenido de **k** ya es 2, indicando que el menor dato se halla en la posición 2 del vector.

Al ejecutar la instrucción 10, incrementa el contenido de **j** en 1, quedando **j** con el valor de 3. Esta nueva situación se presenta en la figura 30.9:

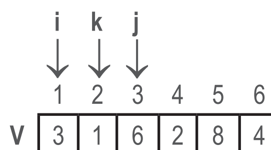


Figura 30.9

Se ejecuta de nuevo la instrucción 7 y compara el dato de la posición 3 con el dato de la posición 2, obteniendo como resultado que la condición es falsa; por lo tanto, no ejecuta la instrucción 8, y continúa en la instrucción 10, es decir, incrementa nuevamente el contenido de **j** en 1, la cual queda valiendo 4. Esta nueva situación se presenta en la figura 30.10:

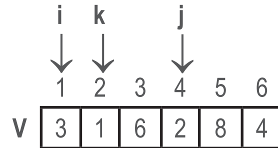


Figura 30.10

Continúa ejecutando las instrucciones del ciclo interno (instrucciones 7 a 10), comparando el contenido de la posición **j** del vector con el contenido de la posición **k**, obteniendo siempre falso como resultado, hasta que el contenido de la variable **j** es 7. Cuando esto sucede, pasa a ejecutar la instrucción 11, en la cual se intercambiará el dato que se halla en la posición **i** con el dato que se halla en la posición **k**. En este momento la situación del vector se presenta como en la figura 30.11:

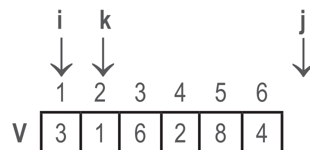


Figura 30.11

Al ejecutar la instrucción 11 el dato que se hallaba en la posición **k** queda en la posición **i**, y el dato que se hallaba en la posición **i** queda en la posición **k**.

Nuestro vector queda como en la figura 30.12:

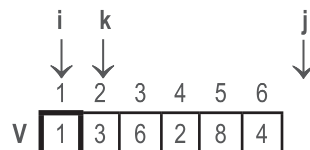


Figura 30.12

El contenido de la posición 1 lo resaltamos indicando que el dato de esa posición ya está ubicado.

Continúa con la ejecución de la instrucción 12, en la cual incrementa el contenido de **i** en 1, y regresa a continuar con el ciclo externo: asignarle nuevamente a **k** el contenido de **i** y comenzar de nuevo la ejecución del ciclo interno, con valor inicial de **j** en 3. La situación en el vector es como en la figura 30.13:

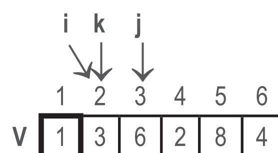


Figura 30.13

Al terminar la ejecución del ciclo interno, por segunda vez, el contenido de la variable **k** será 4, indicando que el menor dato de los que faltan por ordenar, es decir, los datos que hay a partir de la posición 2 del vector, está en la posición 4 del vector. La situación del vector se muestra en la figura 30.14:

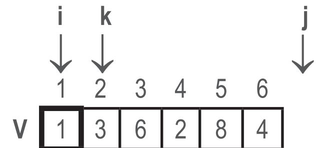


Figura 30.14

Terminada la ejecución del ciclo interno continúa con la instrucción 11 en la cual intercambia el dato que está en la posición **i** con el dato que está en la posición **k** del vector. Es decir, intercambia el dato de la posición 2 con el dato de la posición 4.

La nueva situación del vector, al ejecutar la instrucción 11, se muestra en la figura 30.15:

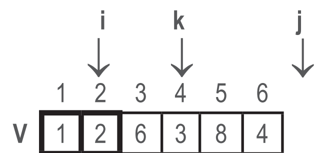


Figura 30.15

Aquí resaltamos el contenido de las dos primeras posiciones indicando que esos dos datos ya están ordenados y en el sitio que les corresponde.

Al ejecutar la instrucción 12 incrementa la **i** en 1, indicando que ya faltan por ordenar sólo los datos a partir de la posición 3, y regresa a ejecutar nuevamente las instrucciones 5 y 6, en las cuales al valor de **k** le asigna el contenido de **i**, y **j** la inicializa en 4. La situación en el vector es como en la figura 30.16:

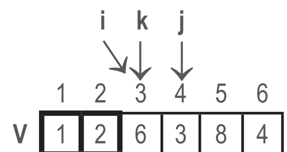


Figura 30.16

El proceso de ordenamiento continuará de esta manera hasta que el valor de **i** sea 6. Cuando el valor de **i** es 6 el proceso termina y los datos del vector están ordenados en forma ascendente. La situación final del vector se muestra en la figura 30.17:

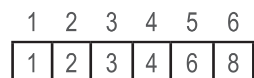


Figura 30.17

Resumen

En este módulo hemos presentado los algoritmos correspondientes a los procesos de búsqueda binaria, que aprovecha el hecho de que los datos se hallan ordenados, y de ordenamiento de los datos de un vector usando el método conocido como método de selección.

Ejercicios propuestos

1. Se tienen dos vectores **A** y **B**, ambos con los datos ordenados ascendentemente. Elabore un subprograma que construya un tercer vector **C** que sea la intercalación de los datos de los vectores **A**, **B** y **C**. En **C** no deben quedar datos repetidos. Los datos del vector **C** deben quedar ordenados ascendentemente a medida que se va construyendo el vector.

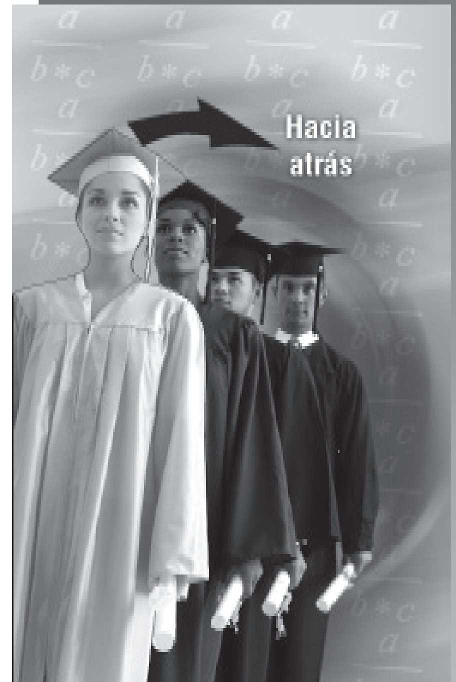
2. Elabore un subprograma que reciba como parámetros dos vectores **A** y **B**, y que construya un tercer vector que sea la intersección de los datos que hay en **A** con los datos que hay en **B**.

3. Elabore un subprograma para ordenar descendentemente los datos de un vector.

4. Dado un número natural **n** de cuatro cifras, elabore un subprograma que encuentre e imprima los números mayor y el menor que se pueden formar con las mismas cifras. Por ejemplo, si **n == 6174**, entonces el número mayor que se puede formar es **7641** y el menor **1467**.

Módulo 31

Operaciones básicas: ordenamiento por burbuja



Introducción

Como ya hemos visto, el hecho de tener los datos ordenados contribuye a que los procesos que se efectúan con los datos de un vector sean más fáciles y eficientes. En el módulo anterior tratamos un método para ordenar los datos de un vector, el cual se conoce como método de selección. Vamos a tratar en este módulo otro método para ordenar los datos de un vector, el cual se denomina por burbuja.

Objetivos del módulo

1. Elaborar un algoritmo para ordenar ascendentemente los datos de un vector aplicando el método burbuja.

Preguntas básicas

1. ¿En qué consiste el método burbuja?
2. ¿Cómo determinar que los datos ya están ordenados antes de terminar el método?

Contenidos del módulo

31.1 Ordenamiento ascendente de los datos de un vector utilizando el método burbuja.

«Para ordenar ascendentemente un conjunto de datos (por ejemplo, las personas de una fila en un banco según el número de transacciones que vayan a efectuar) existen varios métodos. Uno de ellos es identificar a la persona con mayor número de transacciones y ubicarla de última. Luego repetimos este mismo proceso con las que quedaron adelante del último y logramos ubicar el penúltimo. Efectuando este proceso repetidas veces con las personas que faltan por ordenar obtenemos la fila en orden».

Vea en el botón **Una dimensión** del mapa conceptual el video "Módulo 31. Odenamiento por burbuja".

31.1 Ordenamiento ascendente de los datos de un vector utilizando el método burbuja

El método de ordenamiento por burbuja consiste en comparar dos datos consecutivos y ordenarlos ascendentemente, es decir, si el primer dato es mayor que el segundo se intercambian dichos datos, de lo contrario se dejan tal cual están. Cualquiera de las dos situaciones que se hubiera presentado se avanza en el vector para comparar los siguientes dos datos consecutivos. En general, el proceso de comparaciones es: el primer dato con el segundo, el segundo dato con el tercero, el tercer dato con el cuarto, el cuarto dato con el quinto, y así sucesivamente, hasta comparar el penúltimo dato con el último. Como resultado de estas comparaciones, y los intercambios que se hagan, el resultado es que en la última posición quedará el mayor dato en el vector.

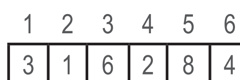
La segunda vez se comparan nuevamente los datos consecutivos, desde el primero con el segundo, hasta que se comparan el antepenúltimo dato con el penúltimo, obteniendo como resultado que el segundo dato mayor queda de penúltimo.

Es decir, en cada pasada de comparaciones, de los datos que faltan por ordenar, se ubica el mayor dato en la posición que le corresponde, o sea de último en el conjunto de datos que faltan por ordenar.

El algoritmo que sigue efectúa esta tarea:

```
1.   void ordenamientoAscendenteBurbuja(V, n)
2.        Variables: i, j: numericaEntera
3.        INICIO
4.             PARA  i  DESDE 1 HASTA   n –1  CON_VARIACION  +1
5.                  PARA  j  DESDE 1 HASTA   n – i  CON_VARIACION  +1
6.                       SI  V[j] > V[j + 1]
7.                            intercambiar(V, j, j + 1)
8.                       Fin(SI)
9.                  Fin(PARA)
10.            Fin(PARA)
11.       FIN
12.  Fin(ordenamientoAscendenteBurbuja)
```

Como ejemplo vamos a considerar el vector de la figura 31.1:



1	2	3	4	5	6
3	1	6	2	8	4

Figura 31.1

Nuestro interés es ordenar los datos de dicho vector en forma ascendente, utilizando el método denominado burbuja.

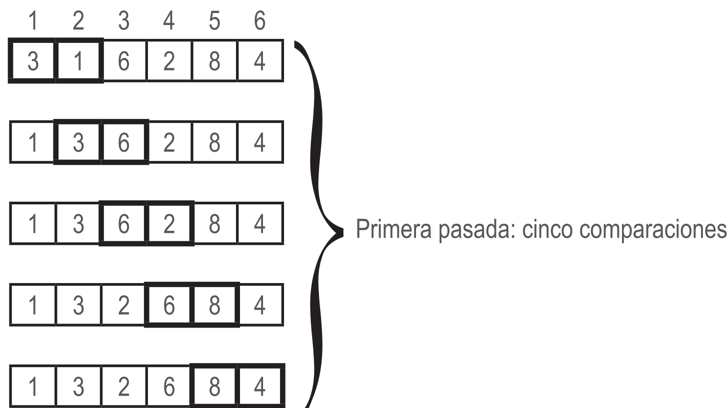


Figura 31.2

En la figura 31.2 se presentan las cinco comparaciones que se efectúan en la primera pasada. La comparación correspondiente a cada pasada se resalta con los bordes más gruesos. A cada comparación la llamaremos **parte** con el fin de evitar redundancias.

En la primera parte se compara el dato de la posición 1 con el dato de la posición 2. Como el dato de la 1 es mayor que el dato de la 2, se intercambian dichos datos, quedando el vector como en la parte 2.

En la parte 2 se compara el dato de la posición 2 con el dato de la posición 3. Como el dato de la 2 es menor que el dato de la 3, los datos se dejan intactos.

En la parte 3 se compara el dato de la posición 3 con el dato de la posición 4. Como el dato de la 3 es mayor que el dato de la 4 se intercambian dichos datos, quedando el vector como en la parte 4.

En la parte 4 se compara el dato de la posición 4 con el dato de la posición 5. Como el dato de la 4 es menor que el dato de la 5, los datos permanecen intactos.

En la parte 5 se compara el dato de la posición 5 con el dato de la posición 6. Como el dato de la 5 es mayor que el dato de la 6, se intercambian dichos datos.

El resultado final de efectuar estas comparaciones y estos intercambios se presenta en la figura 31.3:

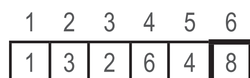


Figura 31.3

En esta figura se ha resaltado el dato de la posición 6, puesto que éste ha quedado en el sitio que le corresponde.

En la segunda pasada se harán nuevamente comparaciones de datos consecutivos desde la posición 1 hasta la posición 5, ya que la 6 ya está en orden. Las comparaciones e intercambios correspondientes a esta segunda pasada se presentan en la figura 31.4:

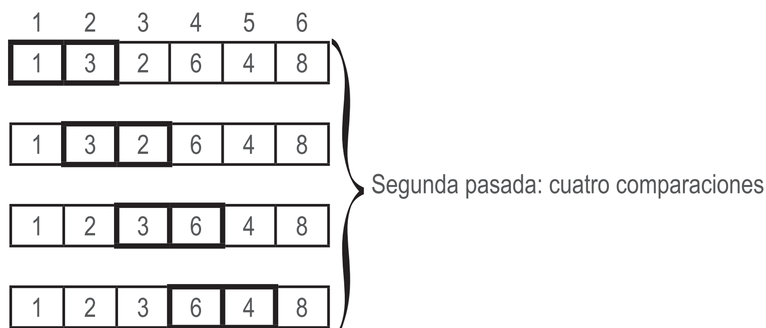


Figura 31.4

Como resultado de esta pasada el vector queda como en la figura 31.5:

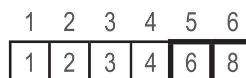


Figura 31.5

En esta figura se resaltan las dos últimas posiciones, indicando que esta parte del vector ya está ordenada.

En la figura 31.6 se muestra cómo es el proceso de comparaciones e intercambios correspondientes a la tercera pasada. Fíjese que las comparaciones sólo se efectúan hasta la posición 4, ya que los datos de las posiciones 5 y 6 ya están ordenados y ubicados en sus correspondientes lugares.

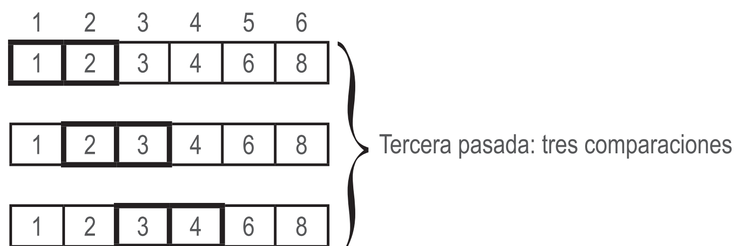


Figura 31.6

En la figura 31.7 se muestra cómo queda el vector al terminar la tercera pasada:

Figura 31.7

En la figura 31.8 se muestran las dos comparaciones correspondientes a la cuarta pasada:

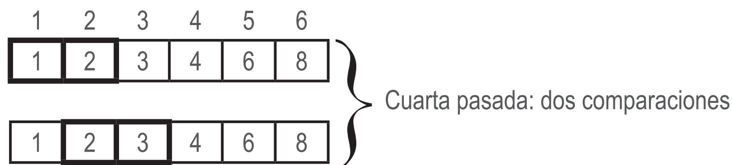


Figura 31.8

En la figura 31.9 se muestra cómo queda el vector al terminar la cuarta pasada:

Figura 31.9

En la figura 31.10 se muestra la comparación que se efectúa en la quinta pasada:

Figura 31.10

En la figura 31.11 se muestra cómo queda el vector al finalizar todo el proceso:

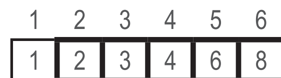


Figura 31.11

Observe que al concluir la tercera pasada los datos del vector ya estaban ordenados. Es decir, las comparaciones correspondientes a las pasadas cuarta y quinta no eran necesarias. Esta situación se puede detectar en el algoritmo. Si utilizamos una variable auxiliar le asignamos un valor antes de comenzar el ciclo interno, y si dentro del ciclo interno ocurrió algún intercambio, modificamos el valor inicial de dicha variable auxiliar. Al terminar el ciclo interno averiguamos cuál es el contenido de dicha variable auxiliar. Si es el mismo que se le asignó inicialmente, significa que no hubo intercambios en el ciclo interno, por consiguiente los datos del vector ya están ordenados y no hay necesidad de hacer más pasadas. Para nuestro ejemplo llamemos a esa variable auxiliar **sw**.

Nuestro algoritmo queda:

```
1.   void ordenamientoAscendenteBurbuja(V, n)
2.        Variables: i, j, sw: numericaEntera
3.        INICIO
4.             PARA  i  DESDE 1 HASTA   n –1  CON_VARIACION  +1
5.                  sw = 0
6.                  PARA  j  DESDE 1 HASTA   n – i  CON_VARIACION  +1
7.                       SI  V[j] > V[j + 1]
8.                            intercambiar(V, j, j + 1)
9.                            sw = 1
10.                      Fin(SI)
```

```
11.                     Fin(PARA)
12.                     SI  sw == 0
13.                          retorne
14.                     Fin(Si)
15.                     Fin(PARA)
16.          FIN
17.   Fin(ordenamientoAscendenteBurbuja)
```

Resumen

En este módulo hemos presentado el algoritmo correspondiente al proceso de ordenar ascendentemente los datos de un vector utilizando el método denominado burbuja.

Ejercicios propuestos

1. Elabore un subprograma que cuente y retorne cuántas veces está cada dato en un vector.

2. Elabore un subprograma que procese un vector de la siguiente manera: el primer dato indica cuántos datos siguientes hay que sumar, el siguiente de los sumados indica lo mismo que el primer dato, y así sucesivamente, hasta recorrer todo el vector. Los resultados de la suma se deben almacenar en un nuevo vector, el cual se debe retornar al programa llamante.

3. Se tienen dos vectores **A** y **B**. En cada uno de ellos se halla almacenado un número entero, con un dígito por posición. Elabore subprogramas para sumar, restar, multiplicar y dividir números enteros representados en esa forma.

4. Elabore un subprograma para determinar e imprimir los números primos desde 1 hasta **n** utilizando el método de Sieve. El método de Sieve consiste en llenar, inicialmente, todos las posiciones del vector con un 1; luego, a partir de la posición 4, colocar en 0 todas las posiciones múltiplas de 2; luego, a partir de la posición 9, colocar todas las posiciones múltiplas de 3 en 0, y así sucesivamente. Al terminar el proceso anterior, las posiciones del vector cuyo contenido sea 1 son los números primos.

5. Una cooperativa de productores de naranjas almacena el total de toneladas cosechadas durante el último año en **n** parcelas (1≤ **n** ≤ 50). En cada parcela se pueden cultivar dos tipos de naranjas: para jugo y para comer. Se conoce el total de toneladas cosechadas de cada uno de los tipos de naranjas. Si en una parcela no se hubiera cosechado alguno de los tipos, entonces habrá 0.

   La información se almacena en un arreglo como se muestra en el siguiente ejemplo:

Naranjas para jugo — Naranjas para comer

1	2	3	4	5	6	. . .	2 * i – 1	2 * i	. . .
100	500	600	0	800	700		500	750	
Parcela 1		Parcela 2		Parcela 3		. . .	Parcela **i**		. . .

- En la parcela 1 se cosecharon 100 toneladas de naranjas para jugo y 500 toneladas de naranjas para comer.
- En la parcela 2 se cosecharon 600 toneladas de naranjas para jugo y 0 toneladas de naranjas para comer.
- En la parcela 3 se cosecharon 800 toneladas de naranjas para jugo y 700 toneladas de naranjas para comer.
- En la parcela **i** se cosecharon 500 toneladas de naranjas para jugo y 750 toneladas de naranjas para comer.

Nota: observe que la información de una misma parcela ocupa posiciones consecutivas en el arreglo.

Se pide elaborar un algoritmo, usando subprogramas, que pueda:

a. Leer la información **n** (1≤ **n** ≤ 50) y las toneladas por tipo de naranja de cada una de las parcelas.
b. Calcular e imprimir el total de la producción por parcela.
c. Eliminar la información de una parcela. El dato requerido para esta opción es el número de parcela a eliminar.
d. Buscar e imprimir el número de una parcela (si hubiera) que no haya tenido producción de ninguno de los tipos de naranjas. Es decir, durante el último año su producción total fue 0.

6. Elabore un algoritmo que genere 9999 enteros, ente 1 y 29, y que imprima cuántas veces aparece el 1, el 2, ..., el 29 entre todos los números que se generaron. Indique al final cuál fue el promedio de esos números.

   La salida tendrá la forma:

   El número 1 apareció ... veces
   El número 2 apareció ... veces
   El número 29 apareció ... veces
   El promedio de ellos fue: ...

Capítulo 7 — Estructura estática de dos dimensiones: matriz

Contenido breve

Módulo 32
Definición, identificación y construcción de una matríz
Módulo 33
Recorridos sobre matrices
Módulo 34
Suma de los datos de filas y columnas de una matriz
Módulo 35
Clases de matrices
Módulo 36
Intercambio de filas y columnas de una matriz
Módulo 37
Ordenamiento de los datos de una matriz con base en los datos de una columna
Módulo 38
Construcción de la transpuesta de una matriz y suma de matrices
Módulo 39
Multiplicación de matrices

Las matrices aparecen por primera vez hacia el año 1850, introducidas por James Joseph Sylvester. El desarrollo inicial de la teoría se debe al matemático William Rowan Hamilton en 1853. En 1858, Arthur Cayley introdujo la notación matricial como una forma abreviada de escribir un sistema de **m** ecuaciones lineales con **n** incógnitas.

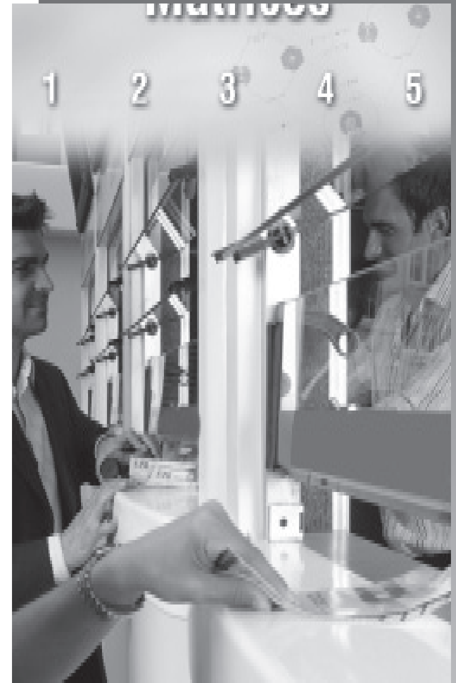
La organización y presentación de datos como estructuras de filas y columnas, en las cuales cada dato se identifica por su fila y columna, facilita el trabajo y la visualización de ellos en muchas situaciones.

Las matrices se utilizan en el cálculo numérico, en la resolución de sistemas de ecuaciones lineales, ecuaciones diferenciales y derivadas parciales. Además de su utilidad para el estudio de sistemas de ecuaciones lineales, las matrices aparecen de forma natural en geometría, estadística, economía, informática, física, etc.

La utilización de matrices (arreglos de dos dimensiones) constituye actualmente una parte esencial de los lenguajes de programación, ya que muchos de los datos que se procesan en un computador se introducen como tablas organizadas por filas y columnas: hojas de cálculo, bases de datos, etc.

Módulo 32

Definición, identificación y construcción de una matriz



«Consideremos una variación a la situación de la fila en un banco planteada en el módulo 26. Allí sólo se contaba con una taquilla. Si tuviéramos cinco taquillas tendríamos cinco filas: una fila por taquilla. Si miramos todas las filas como un conjunto, podemos decir que se tiene un arreglo de dos dimensiones: cinco filas, cada una de ellas con cierta cantidad de clientes. Este arreglo, en el contexto matemático y de elaboración de algoritmos, se conoce como matriz».

Introducción

Las matrices son un recurso de programación simple; en realidad, una matriz puede considerarse como la estructura de datos más simple con la cual se puede trabajar. Presentan la ventaja de que sus elementos son rápidamente accesibles mediante el uso de subíndices, pero tienen la limitación de que son de tamaño fijo. El programador debe definir sus dimensiones al inicio del programa con base en el conocimiento que tenga del problema que va a resolver. En este módulo presentamos una definición de lo que es una matriz, un ejemplo de uso y la forma de construirla.

Objetivos del módulo

1. Definir la estructura matriz.
2. Identificar la estructura matriz.
3. Construir una matriz.

Preguntas básicas

1. ¿Qué es una matriz?
2. ¿Con base en qué se definen las dimensiones de una matriz?
3. ¿Cómo se construye una matriz?

Contenidos del módulo

32.1 Definición de matriz
32.2 Identificación y construcción de una matriz

Vea en el botón **Dos dimensiones** del mapa conceptual el video "Módulo 32. Arreglos de dos dimensiones".

32.1 Definición de matriz

Se llama matriz de orden **m * n** a todo conjunto rectangular de elementos $\mathbf{a_{ij}}$ dispuestos en **m** líneas horizontales (filas) y **n** verticales (columnas) de la forma:

$$\mathbf{A} = \begin{matrix} a_{11} & a_{12} & \dots & a_{1j} & \dots a_{1n} \\ a_{21} & a_{22} & \dots & a_{2j} & \dots a_{2n} \\ \dots & \dots & \dots & \dots & \dots \dots \\ a_{i1} & a_{i2} & \dots & a_{ij} & \dots a_{in} \\ \dots & \dots & \dots & \dots & \dots \dots \\ a_{m1} & a_{m2} & \dots & a_{mj} & \dots a_{mn} \end{matrix}$$

Figura 32.1. Matriz de **m** filas y **n** columnas.

Abreviadamente suele expresarse de la forma $\mathbf{A = (a_{ij})}$ con **i = 1, 2, 3, …, m** y **j = 1, 2, 3, …, n.** Los subíndices indican la posición del elemento dentro de la matriz: el primero **(i)** denota la fila y el segundo **(j)** la columna. Por ejemplo, el elemento a_{46} será el elemento de la fila 4 columna 6.

En la figura 32.1 mostramos un ejemplo de lo que es una matriz.

32.2 Identificación y construcción de una matriz

Con el fin de mostrar la necesidad de utilizar matrices vamos a considerar, de nuevo, un archivo con los datos correspondientes a un censo. En este archivo cada registro contiene la información correspondiente a departamento, municipio, dirección de residencia y número de personas en dicha residencia. Veamos el archivo de la tabla 32.1.

Registro	Departamento	Municipio	Dirección	N.º de personas
1	1	3	-	3
2	3	2	-	1
3	1	1	-	6
4	2	1	-	2
5	2	4	-	8
6	2	3	-	9
7	1	3	-	4
8	EOF			

Tabla 32.1. Archivo de censo.

Si deseamos procesar este archivo con base en lo visto en el módulo 25, y determinar el número de habitantes en cada departamento, debemos definir un vector para cada uno de los **m** departamentos que tenga el país. Dicha situación se muestra en la figura 32.2.

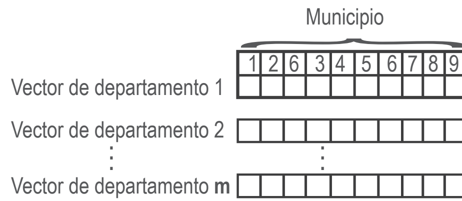


Figura 32.2. Vectores correspondientes a los departamentos.

Manejar **m** vectores tiene los mismos inconvenientes que cuando se manejaban **n** variables para representar el número de habitantes de cada municipio de un departamento. Vamos a adoptar acá una solución análoga.

En vez de definir **m** vectores vamos a considerar una nueva estructura, la cual está conformada con filas y columnas, tal como se muestra en la figura 32.1. Cada fila es como si fuera un vector y la identificaremos con un subíndice.

En nuestro ejemplo cada fila representa un departamento, y las columnas, los municipios correspondientes a cada departamento.

Al procesar completamente el archivo de la tabla 32.1 la matriz que nos proporciona la información acerca de cada municipio de cada departamento se presenta en la figura 32.3.

	mat	1	2	3	4 **n**
	1	6	0	7	0
	2	2	0	9	8
m	3	0	1	0	0

Figura 32.3. Matriz con datos finales del censo.

El programa con el cual hemos construido esta matriz se presenta a continuación:

```
1.  void constMat(mat, m, n)
2.        Variables: depto, mpio, np : numericaEntera
3.                   direc: alfanumérica
4.        INICIO
5.            Inicialice(mat, m, n)
6.            MIENTRAS  not  EOF(censo)
7.                LEA(depto, mpio, direc, np)
8.                mat[depto][mpio] = mat[depto][mpio] + np
9.            Fin(MIENTRAS)
10.           Imprima_por_filas(mat, m, n)
11.       FIN
12. Fin(constMat)
```

En la instrucción 1 se define el título del subprograma con sus parámetros: la matriz **mat** y sus dimensiones **m** y **n**, siendo **m** el número de filas y **n** el número de columnas.

En las instrucciones 2 y 3 definimos las variables con las cuales trabajará nuestro algoritmo: **depto**, para el código del departamento; **mpio**, para el código del municipio; **np**, para el número de personas en cada vivienda; y **direc**, para la dirección de cada residencia.

En la instrucción 5 invocamos un subprograma llamado *inicialice(mat, m, n)*, el cual se encarga de asignar cero a cada una de las celdas de la matriz que se va a trabajar. El algoritmo correspondiente a dicho subprograma lo veremos en el módulo siguiente.

En la instrucción 6 planteamos el ciclo con el cual se procesa el archivo llamado **'censo'**.

En la instrucción 7 leemos los datos correspondientes a cada registro.

En la instrucción 8 actualizamos el acumulado correspondiente al departamento y municipio cuyos códigos se leyeron; es decir, a la celda identificada con la fila **depto** y columna **mpio** se le suma el número de personas que hay en esa dirección.

La instrucción 9 simplemente delimita las instrucciones pertenecientes al ciclo.

En la instrucción 10 invocamos el subprograma *imprimaPorFilas(mat, m, n)***,** el cual produce el informe correspondiente al total de habitantes de cada municipio de cada departamento.

Resumen

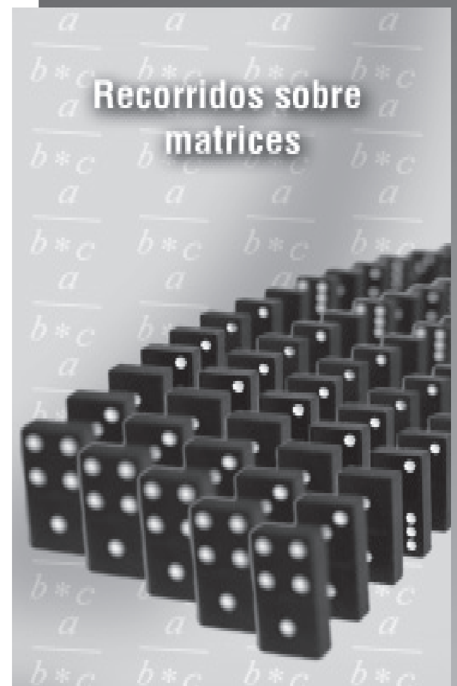
En este módulo hemos introducido el concepto de matriz (estructura estática de dos dimensiones para representar datos) y cómo identificar una matriz, además del algoritmo para construir una matriz con base en datos almacenados en un archivo.

Ejercicio propuesto

1. Haga prueba de escritorio al algoritmo *constMat(mat, m, n)* con los datos de la tabla 32.1 y compruebe que la matriz de la figura 32.3 es correcta.

Módulo 33

Recorridos sobre matrices



Introducción

Cuando se trabaja con matrices una de las operaciones más frecuentes es recorrer la matriz para diferentes propósitos. Algunos de ellos son: asignar valor inicial a cada una de las celdas de la matriz, imprimir el contenido de cada celda recorriendo la matriz por filas e imprimir el contenido de cada celda recorriendo la matriz por columnas. En este módulo nos ocuparemos de elaborar algoritmos (subprogramas) que efectúen dichas tareas.

Objetivos del módulo

1. Realizar la inicialización de los datos de una matriz.
2. Crear un algoritmo para recorrer e imprimir, por filas, los datos de una matriz.
3. Crear un algoritmo para recorrer e imprimir, por columnas, los datos de una matriz.

Preguntas básicas

1. ¿En cuáles situaciones se deben inicializar los datos de una matriz?
2. ¿Cuál subíndice debe variar primero para recorrer una matriz por filas?
3. ¿Cuál subíndice debe variar primero para recorrer una matriz por columnas?

Contenidos del módulo

33.1 Inicialización de los datos de una matriz
33.2 Recorrido e impresión por filas
33.3 Recorrido e impresión por columnas

«Si estamos interesados en determinar cuántos clientes van a efectuar sólo una transacción en el total de las cinco filas en el ejemplo del módulo anterior, necesitamos recorrer todas las filas consultando a cada cliente. Una forma de hacerlo es recorriendo primero una fila, luego la siguiente fila, y así sucesivamente hasta terminar con la quinta. Otra forma es consultando primero al primer cliente de cada fila, luego al segundo cliente de cada fila, luego al tercer cliente de cada fila y así sucesivamente, hasta llegar al último cliente de cada fila».

Vea en el botón **Dos dimensiones** del mapa conceptual el video "Módulo 33. Subprogramas".

33.1 Inicialización de los datos de una matriz

En muchas de las situaciones en las cuales se trabaja con una matriz, ésta se usa como un grupo de contadores o un grupo de acumuladores. En nuestro ejemplo del módulo anterior, la matriz que se utilizó es realmente un conjunto de acumuladores: cada celda tendrá el total de habitantes de cada municipio de cada departamento. Cuando se trabajan acumuladores, tal como lo vimos en el módulo 14, éstos se deben inicializar en cero para poder obtener resultados confiables.

Como en nuestra matriz del censo cada celda es un acumulador, cada celda se debe inicializar en cero. A continuación presentamos un algoritmo con el cual se ejecuta esta tarea:

```
1.  void inicializa(mat, m, n)
2.      Variables: i, j: numericaEntera
3.      INICIO
4.          PARA i DESDE 1 HASTA m CON_VARIACION +1
5.              PARA j DESDE 1 HASTA n CON_VARIACION +1
6.                  mat[i][j] = 0
7.              Fin(PARA)
8.          Fin(PARA)
9.      FIN
10. Fin(inicializa)
```

En la instrucción 1 se define el subprograma con sus parámetros: **mat** es la matriz a la cual se le asignarán valores iniciales a cada una de sus celdas, **m** es el número de filas de la matriz **mat** y **n** es el número de columnas de **mat**.

En la instrucción 2 definimos las variables de trabajo de nuestro algoritmo. Llamamos a estas variables **i** y **j**.

En la instrucción 4 planteamos un ciclo con la variable **i**. Con este ciclo vamos a recorrer las filas de la matriz: cuando el contenido de **i** sea 1 significa que vamos a recorrer la fila 1; cuando el contenido de **i** sea 2 significa que vamos a recorrer la fila 2; cuando el contenido de **i** sea 3 significa que vamos a recorrer la fila 3, y así sucesivamente.

En la instrucción 5 planteamos un ciclo interno, el cual vamos a controlar con la variable **j**. Con este ciclo vamos a recorrer los datos correspondientes a cada fila, es decir, cuando **i** contenga el valor correspondiente a una fila, **j** variará desde 1 hasta **n**.

La forma como varían los contenidos de **i** y **j** para unos valores de **m == 5** y **n == 3** se muestran en la tabla 33.1:

Tabla 33.1. Variación de **i** y **j** con **m** y **n** valiendo 5 y 3, respectivamente.

	i	j
1	1	1
2	1	2
3	1	3
4	2	1
5	2	2

6	2	3
7	3	1
8	3	2
9	3	3
10	4	1
11	4	2
12	4	3
13	5	1
14	5	2
15	5	3

La primera vez que se ejecuta la instrucción 4, a la variable **i** se le asigna el valor de 1 y pasa a ejecutar la instrucción 5, en la cual se le asigna 1 a **j**. Esta situación se ve en la línea 1 de la tabla 33.1.

Al ejecutar la instrucción 6, le asigna 0 a la celda identificada con **i == 1** y **j ==1**.

Al ejecutar la instrucción 7, incrementa el contenido de **j** en 1 y regresa a ejecutar la instrucción 5, en la cual compara el contenido de **j (j == 2)** con el contenido de **n (n == 3)**. Como el contenido de **j** es menor que el contenido de **n**, vuelve a ejecutar la instrucción 6, asignándole 0 a la celda identificada con fila 1 **(i == 1)** y columna 2 **(j == 2)**. Esto se ve en la línea 2 de la tabla 33.1.

Nuevamente ejecuta la instrucción 7: incrementa el contenido de **j** en 1; regresa a la instrucción 5, compara el contenido de **j** con el contenido de **n**, y dependiendo del resultado de la comparación vuelve a ejecutar las instrucciones 6 y 7, o salta y continúa con la instrucción 8.

Cuando ejecuta la instrucción 8, incrementa el contenido de **i** en 1 y regresa a ejecutar de nuevo la instrucción 4 en la cual compara el contenido de la variable **i** con el contenido de la variable **m**. Como el contenido de la variable **i** es menor que el contenido de la variable **m**, vuelve a ejecutar la instrucción 5, es decir, le asigna a **j** nuevamente el valor de 1 y procede a ejecutar nuevamente el ciclo interno, con **j** variando desde 1 hasta **n**. En otras palabras, le asigna 0 a todas las celdas correspondientes a la fila 2. Esto se observa en las líneas 4 a 6 de la tabla 33.1.

Continúa con este proceso hasta que el contenido de la variable **i** sea mayor que el contenido de la variable **m**.

Al terminar de ejecutar los dos ciclos cada una de las celdas de la matriz **mat** tendrá un cero.

33.2 Recorrido e impresión por filas

Cuando se trabaja con matrices una de las tareas más importantes es imprimir el contenido de cada una de las celdas. Para imprimir el contenido de cada una de las celdas de una matriz existen dos formas de hacerlo. Una de ellas es imprimiendo primero el contenido de la fila 1, luego el contenido de la fila 2, luego el de la fila 3 y así sucesivamente. Esta

forma de recorrer e imprimir la matriz se denomina por filas. El algoritmo que sigue ejecuta esta tarea:

```
1.  void imprimePorFilas(mat, m, n)
2.      Variables: i, j: numericaEntera
3.      INICIO
4.          PARA i DESDE 1 HASTA m CON_VARIACION +1
5.              ESCRIBA("fila: ", i)
6.              PARA j DESDE 1 HASTA n CON_VARIACION +1
7.                  ESCRIBA("columna: ", j, " contenido: ", mat[i][j])
8.              Fin(PARA)
9.          Fin(PARA)
10.     FIN
11. Fin(imprimePorFilas)
```

En la instrucción 1 se define el subprograma con sus parámetros: **mat** es la matriz que se desea recorrer e imprimir por filas, **m** es el número de filas de **mat** y **n** es el número de columnas de **mat**.

En la instrucción 2 se definen las variables de trabajo para efectuar el recorrido.

Las instrucciones de ciclo son completamente similares a las desarrolladas en el algoritmo *inicializa(mat, m, n)* del numeral anterior. Realmente, el algoritmo *inicializa(mat, m, n)* lo que hace es inicializar el contenido de cada celda de la matriz, por filas. Es decir, primero asigna ceros a todas las celdas de la fila 1, luego a todas las celdas de la fila 2 y así sucesivamente.

En nuestro algoritmo de recorrido incluimos una nueva instrucción, la instrucción 5, en la cual colocamos el título que indique que se van a imprimir los datos de la fila **i**.

La instrucción 7, que es la instrucción correspondiente al ciclo interno, es aquella en la cual se imprime el contenido de cada celda de una fila **i**. En dicha instrucción imprimimos el mensaje correspondiente a cada columna y el contenido de la celda descrita por **mat[i][j]**.

33.3 Recorrido e impresión por columnas

Cuando se desea recorrer e imprimir la matriz por columnas hay que hacer una "pequeña gran" variación a nuestro algoritmo de recorrido por filas. En el algoritmo que sigue presentamos dicha variación:

```
1.  void imprimePorColumnas(mat, m, n)
2.      Variables: i, j: numericaEntera
3.      INICIO
4.          PARA i DESDE 1 HASTA n CON_VARIACION +1
5.              ESCRIBA("columna: ", i)
6.              PARA j DESDE 1 HASTA m CON_VARIACION +1
7.                  ESCRIBA("fila: ", j, " contenido: ", mat[j][i])
8.              Fin(PARA)
9.          Fin(PARA)
10.     FIN
11. Fin(imprimePorColumnas)
```

En la instrucción 4, que es la instrucción correspondiente al ciclo externo, a la variable controladora del ciclo **i** la ponemos a variar hasta **n**, y no hasta **m**, ya que con esta variable **i** vamos a recorrer las columnas: primero los datos de la columna 1, luego los datos de la columna 2, luego los de la columna 3 y así sucesivamente.

La instrucción 5 produce el mensaje correspondiente a la columna cuyos datos se van a imprimir junto con el número de dicha columna: el contenido de la variable **i**.

En la instrucción 6, que es la instrucción correspondiente al ciclo interno, a la variable controladora **j** la ponemos a variar hasta **m**, y no hasta **n**, ya que con esta variable **j** vamos a recorrer los datos correspondientes a cada columna, y el total de elementos de cada columna es **m**, el número de filas de la matriz.

En la instrucción 7 escribimos cada uno de los datos correspondientes a la columna **i**, es decir, el dato de cada fila **j**: **mat[j][i]**. Observe que en el algoritmo de recorrido por filas se imprime **mat[i][j]** y en el algoritmo de recorrido por columnas se escribe **mat[j][i]**.

Resumen

En este módulo hemos presentado las dos formas principales de recorrer matrices, mostrando como ejemplo los algoritmos correspondientes a inicializar los datos de una matriz y de imprimir los datos de ella, ya sea por filas o por columnas.

Ejercicios propuestos

1. Haga prueba de escritorio al subprograma *imprimePorFilas(mat, m, n)* utilizando la matriz de la figura 32.3, indicando claramente qué escribe.

2. Haga prueba de escritorio al subprograma *imprimePorColumnas(mat, m, n)* utilizando la matriz de la figura 32.3, indicando claramente qué escribe.

3. Se tiene una matriz de **m** filas y **n** columnas, en la cual en cada posición hay un entero entre 1 y 15. Elabore un algoritmo que determine e imprima:

    - Cuántas veces está cada número en la matriz.
    - Cuál es el número que más veces está en la matriz.

4. Se tiene una matriz con **m** filas y **n** columnas en la cual, en cada posición, se halla un dato numérico. Elabore un algoritmo que determine e imprima la sumatoria de los elementos señalados con gris en la matriz.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	. . .
1																.	. . .
2																.	. . .
3																.	. . .
4																.	. . .
5																.	. . .
6																.	. . .
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	. . .
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	. . .
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	. . .

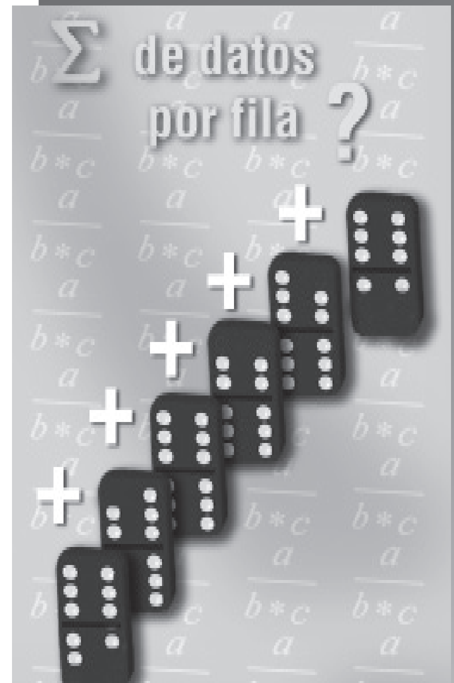
5. Un tablero de damas es un arreglo de ocho filas por ocho columnas. Uno (1) representa la presencia de una ficha roja en el tablero, dos (2) representa la presencia de una ficha negra en el tablero y cero (0) representa la ausencia de ficha. Elabore un algoritmo que determine e imprima:

   - El número de fichas rojas en el tablero.
   - El número de fichas negras en el tablero.
   - El número de espacios vacíos.

   El cálculo de cada uno de los datos anteriores debe efectuarlo con una función que usted debe elaborar e invocar en el algoritmo principal.

Módulo 34

Suma de los datos de filas y columnas de una matriz



Introducción

En el módulo anterior presentamos los algoritmos para recorrer e imprimir una matriz por filas, y otro algoritmo para recorrer e imprimir una matriz por columnas. En general, las operaciones que se necesitan ejecutar con matrices requieren que la matriz se recorra por filas o por columnas. Unas de estas operaciones son totalizar los datos de cada fila y de cada columna. En este módulo presentamos los subprogramas con los cuales se efectúan dichas tareas.

Objetivos del módulo

1. Crear un algoritmo para sumar los datos de cada fila de una matriz.
2. Crear un algoritmo para sumar los datos de cada columna de una matriz.

Preguntas básicas

1. ¿Por qué se requiere sumar los datos de cada fila de una matriz?
2. ¿Por qué se requiere sumar los datos de cada columna de una matriz?

Contenidos del módulo

34.1 Suma de los datos de cada fila de una matriz
34.2 Suma de los datos de cada columna de una matriz

«Un dato que nos podría interesar, teniendo cinco filas en un banco con personas que van a realizar una o más transacciones, es determinar el total de las transacciones que se van a realizar en cada una de las filas. Para efectuar este cálculo debo recorrer cada una de las filas e ir acumulando el número de las transacciones que va a realizar cada cliente».

Vea en el botón **Dos dimensiones** del mapa conceptual el video "Módulo 34. Operaciones sobre matrices".

34.1 Suma de los datos de cada fila de una matriz

Sumar los datos de cada fila es una operación que se requiere con frecuencia. Si consideramos la matriz del ejemplo del censo en varios departamentos, cada fila contiene el número de habitantes de cada municipio de cada departamento. Interesa conocer el total de habitantes de cada departamento: la suma de los datos de las celdas de la fila 1 da el total de habitantes del departamento 1; la suma de los datos de las celdas de la fila 2 da el total de habitantes del departamento 2; la suma de los datos de las celdas de la fila 3 da el total de habitantes del departamento 3, y así sucesivamente.

Para efectuar esta operación se necesita recorrer la matriz por filas, es decir, usaremos las instrucciones de recorrido de la matriz por filas, las cuales fueron explicadas en el algoritmo *imprimePorFilas(mat, m, n)* del módulo anterior.

A continuación presentamos un algoritmo que suma e imprime el total de los datos de las celdas de cada fila de una matriz:

```
1.    void sumaPorFilas(mat, m, n)
2.        Variables: i, j, tf: numericaEntera
3.        INICIO
4.            PARA i DESDE 1 HASTA m CON_VARIACION +1
5.                tf = 0
6.                PARA j DESDE 1 HASTA n CON_VARIACION +1
7.                    tf = tf + mat[i][j]
8.                Fin(PARA)
9.                ESCRIBA("total fila: ", i, " es: ", tf)
10.           Fin(PARA)
11.       FIN
12.   Fin(sumaPorFilas)
```

La instrucción 1 define el subprograma con sus parámetros: **mat**, la matriz en la cual hay que efectuar el proceso de suma de los datos de cada fila; **m**, el número de filas de **mat**; y **n,** el número de columnas de **mat**.

En la instrucción 2 definimos las variables de trabajo para ejecutar la tarea de sumar e imprimir el total de los datos de las celdas de cada fila: **i**, para recorrer las filas; **j**, para recorrer las celdas de cada fila; y **tf**, para llevar el acumulado de los datos de las celdas de la fila **i**.

En la instrucción 4 planteamos el ciclo de variación de **i**, para recorrer la matriz por filas.

En la instrucción 5 se inicializa el acumulador de los datos de cada fila en cero. Este acumulador, **tf**, se debe inicializar en cero cada vez que se cambie de fila, ya que si no hacemos esto, cuando se cambie de fila arrastrará el acumulado de la fila anterior al de la nueva fila.

En la instrucción 6 se plantea el ciclo para recorrer las celdas de cada fila.

En la instrucción 7 se acumula el contenido de la celda **mat[i][j]** (fila **i**, columna **j**) en la variable **tf**.

La instrucción 8 delimita la ejecución del ciclo interno.

En la instrucción 9 se imprime el total de la suma de los datos de las celdas de la fila **i**, con su correspondiente mensaje.

La instrucción 10 delimita la terminación del ciclo externo.

Como ejemplo, consideremos la matriz de la figura 34.1:

								n
		1	2	3	4	5	6	7
	1	3	1	6	2	8	4	5
	2	9	7	7	4	1	2	4
	3	5	6	2	8	5	4	7
	4	3	5	5	4	8	9	3
	5	8	4	5	8	8	7	6
	6	6	3	5	9	8	4	2
m	7	2	5	7	5	4	4	1

Figura 34.1

Al ejecutar por primera vez la instrucción 4, la variable **i** toma el valor de 1, significando que vamos a recorrer la fila 1.

En la instrucción 5 se inicializa el contenido de **tf** en cero.

Al ejecutar por primera vez la instrucción 6, la variable **j** toma el valor de 1. En este instante, el contenido de las variables **i** y **j** es 1.

Al ejecutar la instrucción 7 le suma a **tf** el contenido de la celda de la fila 1 columna 1, es decir, 3. El contenido de **tf** será 3.

Al ejecutar la instrucción 8, el fin del PARA interno, incrementa el contenido de **j** en 1. La **j** queda valiendo 2, mientras que la **i** sigue en 1.

Ejecuta de nuevo la instrucción 7, en la cual le suma el contenido de la celda fila 1 columna 2, cuyo valor es 1, a la variable **tf**. El contenido de **tf** será 4.

De esta forma continúa ejecutando el ciclo interno hasta que el contenido de la variable **j** sea mayor que 7. Cuando esto sucede, continúa con la instrucción 9, en la cual imprime el contenido de **tf**, que para nuestro ejemplo es 29.

Luego de escribir el resultado de la suma de la primera fila, llega a la instrucción 10, que es el fin del PARA externo. En esta instrucción incrementa automáticamente el contenido de la **i** en 1 y regresa a la instrucción 4 a comparar el contenido de la **i** con el contenido de la **n**. Si el contenido de la **i** es menor o igual que el contenido de la **n** ejecuta nuevamente las instrucciones 5 a 10.

A la variable **tf** le asigna nuevamente cero y recorre la fila **i** sumando el contenido de cada una de las celdas con el ciclo interno y luego imprimiendo el contenido de **tf**.

Al terminar de ejecutar el ciclo externo finaliza la ejecución del subprograma habiendo escrito la suma de los datos de las celdas de cada fila.

34.2 Suma de los datos de cada columna de una matriz

Así como en muchas situaciones es necesario sumar los datos de las celdas de cada fila, hay otras situaciones en las cuales lo que se requiere es sumar los datos de las celdas de cada columna. Para ello haremos uso del algoritmo *imprimePorColumnas(mat, m, n)* del módulo anterior.

Para totalizar los datos de las celdas de cada columna de una matriz, basta con plantear los ciclos de recorrido por columnas e insertar las instrucciones propias para manejar el acumulador y producir el informe de total de cada columna. Dicho algoritmo se presenta a continuación:

```
1.   void sumaPorColumnas(mat, m, n)
2.       Variables: i, j, tc: numericaEntera
3.       INICIO
4.           PARA  i  DESDE  1  HASTA  n  CON_VARIACION  +1
5.               tc = 0
6.               PARA  j  DESDE  1  HASTA  m  CON_VARIACION  +1
7.                   tc = tc + mat[j][i]
8.               Fin(PARA)
9.               ESCRIBA(“total columna: “, i, “ es: “, tc)
10.          Fin(PARA)
11.      FIN
12.  Fin(sumaPorColumnas)
```

La instrucción 1 define el subprograma con sus parámetros: **mat**, la matriz en la cual hay que efectuar el proceso de suma de los datos de cada columna; **m**, el número de filas de **mat;** y **n,** el número de columnas de **mat**.

En la instrucción 2 definimos las variables de trabajo para ejecutar la tarea de sumar e imprimir el total de los datos de las celdas de cada columna: **i**, para recorrer las columnas; **j**, para recorrer las celdas de cada columna; y **tf**, para llevar el acumulado de los datos de las celdas de la columna **i**.

En la instrucción 4 planteamos el ciclo de variación de la **i**, para recorrer la matriz por columnas.

En la instrucción 5 se inicializa el acumulador de los datos de cada columna en cero. Este acumulador, **tc**, se debe inicializar en cero cada vez que se cambie de columna, ya que si no hacemos esto, cuando se cambie de columna arrastrará el acumulado de la columna anterior al de la nueva columna.

En la instrucción 6 se plantea el ciclo para recorrer las celdas de cada columna.

En la instrucción 7 se acumula el contenido de la celda **mat[j][i]** (fila **j**, columna **i**) en la variable **tc**.

La instrucción 8 delimita la ejecución del ciclo interno.

En la instrucción 9 se imprime el total de la suma de los datos de las celdas de la columna **i**, con su correspondiente mensaje.

La instrucción 10 delimita la terminación del ciclo externo.

Resumen

En este módulo hemos tratado dos subprogramas de aplicación de recorrido sobre matrices: sumar e imprimir el resultado de los datos de las celdas de cada fila de una matriz y sumar e imprimir el resultado de los datos de las celdas de cada columna de una matriz.

Ejercicios propuestos

1. Haga prueba de escritorio al subprograma *sumaPorFilas(mat, m, n)* con la matriz de la figura 34.1, indicando claramente qué imprime.

2. Haga prueba de escritorio al subprograma *sumaPorColumnas(mat, m, n)* con la matriz de la figura 34.1, indicando claramente qué imprime.

3. Se tiene una matriz con **m** filas y **n** columnas en la cual, en cada posición, se halla un dato numérico. Elabore un algoritmo que construya dos vectores así: uno con la sumatoria de los elementos de cada fila y otro con la sumatoria de los elementos de cada columna. Elabore funciones para construir dichos vectores, las cuales debe invocar en su algoritmo principal. En el primer vector construido determine la fila cuya sumatoria es mayor. En el segundo vector determine la columna cuya sumatoria es menor. Para determinar el mayor y el menor elabore también funciones, las cuales también debe invocar en su algoritmo principal.

4. Se tiene una matriz de **m** filas y **n** columnas donde cada elemento de la matriz representa las ventas correspondientes a cada uno de los **m** vendedores de una empresa, para cada uno de los **n** años que ha tenido de operación. Elabore un algoritmo que construya dos vectores así: uno con el total de ventas de cada vendedor en los **n** años, y el otro con el total de ventas de cada año. La construcción de los vectores la debe efectuar con funciones que usted debe elaborar e invocar desde el algoritmo principal. Luego, con los vectores construidos, determine el vendedor que más ventas ha efectuado y el año en que menos se vendió. Haga estas últimas dos operaciones con funciones que usted elabore.

5. Elabore un algoritmo que forme e imprima una matriz con las siguientes características:

    - La primera fila y la primera columna tienen como elementos los números del 0 al 20.
    - Los demás elementos se obtienen de multiplicar cada elemento de la fila 1 por cada elemento de la columna 1. Ejemplo:

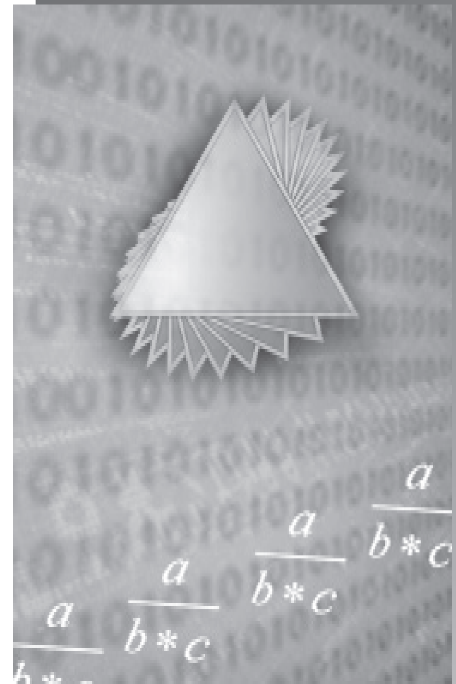
```
0  1  2  3  4   5   6   7   8   9  10  11  12  13  14  15  16 ...
1  1  2  3  4   5   6   7   8   9  10  11  12  13  14  15  16 ...
2  2  4  6  8  10  12  14  16  18  20  22  24  26  28  30  32 ...
3  3  6  9  12 15  18  21  24  27  30  33  36  39  42  45  48 ...
.
.
.
```

Para construir los datos de cada fila elabore un subprograma que debe invocar en el algoritmo principal. Al final imprima, por filas, la matriz creada. La impresión de la matriz debe efectuarla con un subprograma que usted debe elaborar e invocar en el algoritmo principal.

6. Considerando la matriz del ejercicio 4 del módulo 33 elabore un algoritmo que construya y retorne dos vectores: uno con la suma de los datos de las posiciones sin sombrear de cada fila, y otro con la suma de los datos de las posiciones sombreadas de cada columna.

Módulo 35

Clases de matrices



Introducción

Cuando se trabaja con matrices se presentan diferentes situaciones, las cuales afectan sus características. Según estas características, se ha establecido cierta clasificación de las matrices. De acuerdo con esta clasificación se podrán efectuar, o no, determinadas operaciones sobre las matrices. Nos ocuparemos en este módulo de algunos tipos de matrices y de algunas operaciones sobre ellas.

Objetivos del módulo

1. Identificar diferentes clases de matrices.
2. Crear un algoritmo para sumar los elementos de la diagonal principal de una matriz cuadrada.
3. Crear un algoritmo para sumar los elementos de la diagonal secundaria de una matriz cuadrada.

Preguntas básicas

1. ¿Qué es una matriz cuadrada?
2. ¿Qué es una matriz rectangular?
3. ¿Cuáles elementos conforman la diagonal principal de una matriz cuadrada?

Contenidos del módulo

35.1 Clasificación de matrices
35.2 Suma de los elementos de la diagonal principal de una matriz cuadrada
35.3 Suma de los elementos de la diagonal secundaria de una matriz cuadrada

«Una matriz es un modelo matemático con múltiples aplicaciones. Dependiendo de la aplicación se presentan diferentes tipos de matrices. En la solución de un sistema de ecuaciones la matriz resultante es triangular; en la representación de relaciones la matriz que resulta es, por lo general, una matriz dispersa».

Vea en el botón **Dos dimensiones** del mapa conceptual el video "Módulo 35. Variaciones con matrices según su tamaño".

35.1 Clasificación de matrices

Las matrices se clasifican según la forma y según algunas características con respecto a la distribución de sus datos.

Según la forma, las matrices pueden ser *rectangulares* o *cuadradas*. Las rectangulares son aquellas en las que el número de filas es diferente del número de columnas, y las cuadradas son aquellas en las que el número de filas es igual al número de columnas.

Según la distribución de sus datos, existen muchas clases de matrices. Entre ellas podemos destacar las siguientes:

- Matriz *simétrica*: es una matriz cuadrada en la que el dato de una celda de la fila **i**, columna **j,** es igual al dato de la celda de la fila **j**, columna **i**.
- Matriz *identidad*: es una matriz cuadrada en la que todos los elementos son 0, excepto los de la diagonal principal, que son 1. Los elementos de la diagonal principal son aquellos en los cuales la fila es igual a la columna.
- Matriz *triangular inferior*: es una matriz cuadrada en la cual los únicos elementos diferentes de cero son los de la diagonal principal y los que están debajo de ella.
- Matriz *triangular superior*: es una matriz cuadrada en la cual los únicos elementos diferentes de cero son los de la diagonal principal y los que están por encima de ella.
- Matriz *diagonal*: es una matriz cuadrada en la que los únicos elementos diferentes de cero son los de la diagonal principal.

Según la clase de matriz que se esté trabajando, las operaciones sobre ellas varían. Por ejemplo, para calcular el determinante de una matriz, ésta debe ser cuadrada.

35.2 Suma de los elementos de la diagonal principal de una matriz cuadrada

Como un ejemplo de trabajo con matrices cuadradas presentamos un algoritmo con el que se suman los elementos de su diagonal principal. El siguiente es un subprograma que efectúa dicha tarea:

```
1.  entero sumaDiagPpal(mat, n)
2.      Variables: i, s: numericaEntera
3.      INICIO
4.          s = 0
5.          PARA i DESDE 1 HASTA n CON_VARIACION +1
6.              s = s + mat[i][i]
7.          Fin(PARA)
8.          retorne(s)
9.      FIN
10. Fin(sumaDiagPpal)
```

En la instrucción 1 se define el subprograma, el cual retornará un dato entero, con sus parámetros: **mat**, la matriz a la cual le vamos a sumar los elementos de la diagonal principal, y **n,** el número de filas y columnas de la matriz **mat**.

En la instrucción 2 definimos nuestras variables de trabajo: **i**, variable con la cual recorremos los elementos de la diagonal principal, y **s**, la variable en la cual almacenamos la suma de dichos elementos.

En la instrucción 4 se inicializa el valor del acumulador **s** en cero.

En la instrucción 5 planteamos el ciclo con el cual se recorren los elementos de la diagonal principal. Como ya habíamos dicho, los elementos de la diagonal principal son aquellos en los cuales la fila es igual a la columna; por lo tanto, sólo requerimos una variable para recorrerlos.

En la instrucción 6 acumulamos en la variable **s** dichos elementos.

La instrucción 7 delimita las instrucciones del ciclo.

En la instrucción 8 se retorna el valor de **s**.

Como ejemplo, consideremos la matriz de la figura 35.1:

								n
		1	2	3	4	5	6	7
	1	3	1	6	2	8	4	5
	2	9	7	7	4	1	2	4
	3	5	6	2	8	5	4	7
	4	3	5	5	4	8	9	3
	5	8	4	5	8	8	7	6
	6	6	3	5	9	8	4	2
n =	7	2	5	7	5	4	4	1

Figura 35.1

Al ejecutar nuestro subprograma *sumaDiagPpal(mat, n)* retorna un valor de 29.

35.3 Suma de los elementos de la diagonal secundaria de una matriz cuadrada

Como un segundo ejemplo, vamos a construir un algoritmo con el que se sumen los elementos de la diagonal secundaria de una matriz cuadrada. Una celda pertenece a la diagonal secundaria si la suma de la fila y la columna que identifican la celda da **n + 1**, siendo **n** el número de filas y columnas de la matriz. El algoritmo que sigue ejecuta dicha tarea:

```
1.  entero sumaDiagSec(mat, n)(1)
2.      Variables: i, j, s: numericaEntera
3.      INICIO
4.          s = 0
5.          j = n
6.          PARA  i  DESDE  1  HASTA  n  CON_VARIACION  +1
7.              s = s + mat[i][j]
8.              j = j – 1
9.          Fin(PARA)
10.         retorne(s)
11.     FIN
12. Fin(sumaDiagSec)
```

En la instrucción 1 definimos el subprograma, el cual retornará un valor entero, con sus parámetros: **mat**, la matriz sobre la cual se va a trabajar, y **n,** el orden de la matriz.

En la instrucción 2 se definen las variables de trabajo: **i**, para identificar la fila de una celda de la diagonal secundaria; **j,** para identificar la columna de una celda de la diagonal secundaria; y **s**, la variable en la cual llevaremos la suma de los elementos de la diagonal secundaria.

En la instrucción 4 inicializamos el acumulador en cero.

En la instrucción 5 inicializamos el valor de **j** en **n**, es decir, la última columna. La variable **j** la utilizaremos para identificar la columna de un elemento perteneciente a la diagonal secundaria.

En la instrucción 6 planteamos el ciclo con el cual recorreremos la matriz. La variable controladora del ciclo, es decir la **i**, la utilizaremos para identificar la fila de un elemento de la diagonal secundaria.

En la instrucción 7 acumulamos en la variable **s** el dato de una celda perteneciente a la diagonal secundaria. Dicha celda la identificamos con la fila **i** y la columna **j**.

En la instrucción 8 le restamos 1 a **j**, la variable con la cual se identifica la columna de un elemento de la diagonal secundaria.

La instrucción 9 delimita las instrucciones del ciclo.

En la instrucción 10 se retorna el contenido de la variable **s**, la cual contiene la suma de los elementos de la diagonal secundaria.

Como ejemplo consideremos la matriz de la figura 35.2.

		1	2	3	4	5	6	**n** 7
	1	3	1	6	2	8	4	5
	2	9	7	7	4	1	2	4
	3	5	6	2	8	5	4	7
	4	3	5	5	4	8	9	3
	5	8	4	5	8	8	7	6
	6	6	3	5	9	8	4	2
n =	7	2	5	7	5	4	4	1

Figura 35.2

Al ejecutar nuestro algoritmo *sumaDiagSec(mat, n)* retorna un valor de 26.

En el algoritmo anterior utilizamos dos variables para identificar las celdas pertenecientes a la diagonal secundaria: la variable **i**, con la cual se identifica la fila de una celda perteneciente a la diagonal secundaria, y la variable **j**, con la cual se identifica la columna de una celda perteneciente a la diagonal secundaria. Fíjese que para nuestro ejemplo de la figura 35.2, el valor inicial de **i** es 1, mientras que el valor inicial de **j** es 7. Con estos valores identificamos el primer elemento de la diagonal secundaria: la celda de la fila 1 columna 7. El

segundo elemento de la diagonal secundaria se halla en la celda cuya ubicación es la fila 2 columna 6. En la instrucción 8, que está dentro del ciclo, le restamos 1 a **j**, y cuando llega al fin del PARA, automáticamente incrementa el valor de **i** en 1. De esta manera **i** queda valiendo 2 y **j** 6. El siguiente elemento de la diagonal secundaria es el de la fila 3 columna 5. Como ya vimos, dentro del ciclo disminuye el valor de **j** en 1, y con la instrucción Fin(PARA) se incrementa el valor de **i** en 1. Esa es la forma como variamos a **i** y a **j** para generar los subíndices con los cuales se identifican los elementos de la diagonal secundaria.

Si consideramos que la suma de los subíndices de las celdas que identifican los elementos de la diagonal secundaria es **n + 1**, podemos escribir otro algoritmo para calcular dicha suma, sin necesidad de utilizar la variable **j**.

Veamos: **i + j = n + 1**

Despejando **j** obtenemos: **j = n – i + 1**

En el siguiente algoritmo usamos esta propiedad:

```
1.    entero sumaDiagSec(mat, n)(2)
2.        Variables: i, s: numericaEntera
3.          INICIO
4.              s = 0
5.              PARA  i  DESDE  1  HASTA  n  CON_VARIACION  +1
6.                    s = s + mat[i][n – i + 1]
7.              Fin(PARA)
8.             retorne(s)
9.        FIN
10.   Fin(sumaDiagSec)
```

Dejamos al lector la tarea de comprobar el funcionamiento de este algoritmo.

Resumen

En este módulo hemos presentado algunas de las principales clases de matrices que están definidas, mostrando cómo, según la clase de matriz, hay algunas operaciones que se pueden efectuar sobre ellas y otras no. Mostramos como ejemplo la suma de las diagonales principal y secundaria de una matriz cuadrada.

Ejercicios propuestos

1. Elabore un subprograma que calcule y retorne la suma de los elementos que están por encima de la diagonal principal en una matriz cuadrada de orden **n**.

2. Elabore un subprograma que calcule y retorne la suma de los elementos que están por debajo de la diagonal principal en una matriz cuadrada de orden **n**.

3. Elabore un subprograma que calcule y retorne la suma de los elementos que están por encima de la diagonal secundaria en una matriz cuadrada de orden **n**.

4. Elabore un subprograma que calcule y retorne la suma de los elementos que están por debajo de la diagonal secundaria en una matriz cuadrada de orden **n**.

5. Elabore un subprograma que determine si una matriz cuadrada entrada como parámetro es simétrica o no. Su subprograma debe retornar 1 si es simétrica, 0 de lo contrario.

6. Elabore un subprograma que determine si una matriz cuadrada entrada como parámetro es triangular inferior o no. Su subprograma debe retornar 1 si es triangular inferior, 0 de lo contrario.

7. Elabore un subprograma que determine si una matriz cuadrada entrada como parámetro es triangular superior o no. Su subprograma debe retornar 1 si es triangular superior, 0 de lo contrario.

8. Elabore un subprograma que determine si una matriz cuadrada entrada como parámetro es matriz identidad o no. Su subprograma debe retornar 1 si es matriz identidad, 0 de lo contrario.

9. Elabore un subprograma que determine si una matriz cuadrada entrada como parámetro es diagonal o no. Su subprograma debe retornar 1 si es diagonal, 0 de lo contrario.

Módulo 36

Intercambio de filas y columnas de una matriz

«Consideremos de nuevo la situación de las filas de clientes que van a ser atendidos en las diferentes taquillas de un banco, y supongamos que se tiene una fila en la cual se está atendiendo el pago de servicios públicos y otra en la cual se atiende a quienes van a efectuar retiros con libreta. Si la máquina en la cual se atienden los pagos de servicios sufre algún percance, y la máquina en la cual se está atendiendo a los clientes que retiran con libreta sigue funcionando correctamente, habrá que intercambiar a los clientes de ambas filas para no perjudicarlos».

Introducción

Dentro de las operaciones que se efectúan con matrices es necesario, en muchas de ellas, intercambiar el contenido de las celdas de dos filas o de dos columnas. En este módulo presentamos los subprogramas correspondientes a estas tareas.

Objetivos del módulo

1. Crear un algoritmo para intercambiar los datos de dos filas dadas en una matriz.
2. Crear un algoritmo para intercambiar los datos de dos columnas dadas en una matriz.

Preguntas básicas

1. ¿En qué consiste intercambiar los datos de dos filas de una matriz?
2. ¿En qué consiste intercambiar los datos de dos columnas de una matriz?
3. ¿Para qué se necesita intercambiar los datos de dos filas en una matriz?

Contenidos del módulo

36.1 Intercambio de dos filas
36.2 Intercambio de dos columnas

Vea en el botón **Dos dimensiones** del mapa conceptual el video "Módulo 36. Ejercicios con arreglos de dos dimensiones".

36.1 Intercambio de dos filas

Otra de las operaciones que se requieren con frecuencia cuando se manipulan matrices es intercambiar los datos correspondientes a dos filas dadas. El algoritmo que sigue efectúa dicha tarea:

```
1.  void intercambiaFilas(mat, m, n, i, j)   // mat: parámetro por referencia
2.        Variables: k, aux: numericaEntera
3.        INICIO
4.            PARA  k  DESDE  1  HASTA  n  CON_VARIACION  +1
5.                aux = mat[i][k]
6.                mat[i][k] = mat[j][k]
7.                mat[j][k] = aux
8.            Fin(PARA)
9.        FIN
10. Fin(intercambiaFilas)
```

En la instrucción 1 definimos el subprograma con sus parámetros: **mat**, la matriz en la cual se efectúa el intercambio; **m**, el número de filas de la matriz **mat**; **n**, el número de columnas de la matriz **mat**; **i** y **j**, las filas cuyos datos hay que intercambiar. El parámetro **mat** es por referencia.

En la instrucción 2 definimos las variables de trabajo: **k**, para recorrer las filas cuyos datos se desean intercambiar, y **aux**, variable auxiliar para poder efectuar el intercambio.

En la instrucción 4 planteamos el ciclo con el cual se recorren simultáneamente las filas **i** y **j**.

Las instrucciones 5 a 7 son las instrucciones con las cuales se intercambia el dato de la celda de la fila **i** columna **k**, con el dato de la celda de la fila **j** columna **k**. En la instrucción 5 se guarda en la variable auxiliar **aux** el contenido de la celda fila **i** columna **k**; en la instrucción 6 trasladamos el dato de la celda de la fila **j** columna **k** hacia la celda fila **i** columna **k**, y en la instrucción 7 llevamos lo que tenemos en la variable auxiliar **aux** (lo que había en la celda fila **i** columna **k**) hacia la celda de la fila **j** columna **k**.

Las instrucciones del ciclo se ejecutan hasta que el contenido de la variable **k** sea mayor que **n**. Cuando esto sucede habrá intercambiado los datos de la fila **i** con los datos de la fila **j**.

Como ejemplo, observe la figura 36.1. En 36.1*a* está la matriz antes de efectuar el intercambio de los datos de la fila 2 con los datos de la fila 5, y en 36.1*b* está la matriz después de efectuar el intercambio de los datos de la fila 2 con la fila 5.

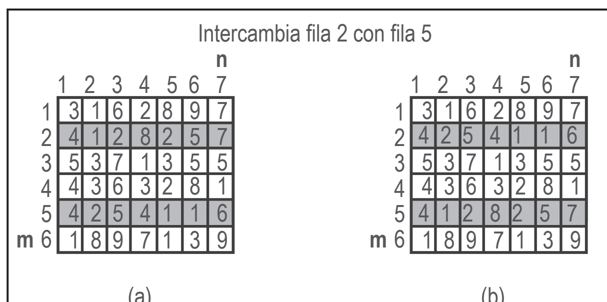


Figura 36.1. Intercambio de dos filas.

36.2 Intercambio de dos columnas

Así como intercambiar filas es una operación que se requiere con cierta frecuencia en la manipulación de matrices, intercambiar columnas también. El proceso de intercambiar columnas es similar al proceso de intercambiar filas. La diferencia básica es que en vez de recorrer las filas se recorren las columnas cuyos datos hay que intercambiar. El siguiente algoritmo efectúa dicho proceso:

```
1.   void intercambiaColumnas(mat, m, n, i, j)     // mat: parámetro por referencia
2.        Variables: k, aux: numericaEntera
3.        INICIO
4.             PARA k DESDE 1 HASTA m CON_VARIACION +1
5.                  aux = mat[k][i]
6.                  mat[k][i] = mat[k][j]
7.                  mat[k][j] = aux
8.             Fin(PARA)
9.        FIN
10.  Fin(intercambiaColumnas)
```

Como ejemplo presentamos la matriz de la figura 36.2, en la cual se intercambian los datos de las celdas de la columna 2 con los datos de las celdas de la columna 5.

Intercambia columna 2 con columna 5

(a)

	1	2	3	4	5	6	7 (n)
1	3	1	6	2	8	9	7
2	4	1	2	8	2	5	7
3	5	3	7	1	3	5	5
4	4	3	6	3	2	8	1
5	4	2	5	4	1	1	6
m 6	1	8	9	7	1	3	9

(b)

	1	2	3	4	5	6	7 (n)
1	3	8	6	2	1	9	7
2	4	2	2	8	1	5	7
3	5	3	7	1	3	5	5
4	4	2	6	3	3	8	1
5	4	1	5	4	2	1	6
m 6	1	1	9	7	8	3	9

Figura 36.2. Intercambio de dos columnas.

Resumen

En este módulo hemos presentado los algoritmos correspondientes a intercambiar los datos de dos filas y de dos columnas de una matriz. Este es un proceso que se requiere con mucha frecuencia, con diferentes propósitos, en la manipulación de matrices.

Ejercicios propuestos

1. Se tiene una matriz de **m** filas y **n** columnas. Elabore un algoritmo que intercambie los elementos de cada fila así: el dato de la primera columna con el dato de la última columna; el dato de la segunda columna con el dato de la penúltima columna; el dato de la tercera columna con el dato de la antepenúltima columna, y así sucesivamente. El proceso de intercambio debe efectuarlo con un subprograma que usted debe elaborar e invocar en su algoritmo principal.

2. Un tablero de damas es un arreglo de ocho filas por ocho columnas. Uno (1) representa la presencia de una ficha roja en el tablero, dos (2) representa la presencia de una ficha negra en el tablero y cero (0) representa la ausencia de ficha. Elabore un algoritmo que intercambie las fichas rojas con las fichas negras. El intercambio se hace recorriendo la matriz por filas e intercambiando así: la primera ficha roja que encuentre con la primera ficha negra que encuentre, la segunda ficha roja que encuentre con la segunda ficha negra que encuentre, y así sucesivamente. Tenga en cuenta que puede haber más fichas de un color que de otro.

3. Elabore un algoritmo que lea una matriz de **n * m** elementos y un valor. Luego modifique la matriz multiplicando cada elemento por el valor leído inicialmente. La modificación de la matriz debe efectuarla con un subprograma que usted debe elaborar e invocar en el algoritmo principal.

4. Dada una matriz de **m** filas y **n** columnas, la cual se halla ordenada en forma ascendente por los datos de la primera columna, elabore un algoritmo que inserte una fila en dicha matriz sin dañar el ordenamiento por la primera columna. Las operaciones para determinar la posición en la cual debe quedar la fila a insertar y el proceso de inserción debe efectuarlas con subprogramas que usted debe elaborar e invocar en el algoritmo principal.

Módulo 37

Ordenamiento de los datos de una matriz con base en los datos de una columna

Introducción

Tal como lo hemos venido planteando en los módulos anteriores, existe un sinnúmero de operaciones que se deben efectuar cuando se manipulan matrices. Otra de ellas es tener que ordenar los datos de una matriz tomando como base los datos de una columna. En este módulo se presenta un algoritmo que efectúa dicha tarea.

Objetivos del módulo

1. Crear un algoritmo que ordena una matriz con base en los datos de una columna.

Preguntas básicas

1. ¿En qué consiste ordenar una matriz por una columna?
2. ¿Qué implica intercambiar dos datos de una columna?

Contenidos del módulo

37.1 Ordenar los datos de una matriz con base en los datos de una columna

«Si utilizamos una matriz para representar las ventas de cada producto en los diferentes días del mes en un almacén, y en la primera columna de la matriz tenemos el código de cada producto, nos interesará tener ordenados los datos de acuerdo al código del producto. Esto implica que los datos de la matriz deberán ser ordenados de acuerdo al dato de la primera columna».

Vea en el botón **Dos dimensiones** del mapa conceptual el video "Módulo 37. Operaciones con matrices".

37.1 Ordenar los datos de una matriz con base en los datos de una columna

Sucede con frecuencia que cuando se manejan datos en una matriz, en una columna se maneja, digamos, un código, y en el resto de la fila se manejan datos correspondientes a ese código.

Supongamos que en cada fila se manejan los siguientes datos: el código de un artículo y los datos correspondientes a las ventas de ese artículo en cada uno de los almacenes que maneja la compañía.

Si miramos la matriz de la figura 37.1*a*, el dato de la columna 1 fila 1 es el código de un artículo (artículo con código 3) y los demás datos de la fila 1 son las ventas del artículo 3, es decir, 1, 4, 9, 7, 7, 5 y 6.

El dato de la fila 2 columna 1 es el código de un artículo (artículo con código 1) y los demás datos de la fila 2 son las ventas del artículo 1, es decir, 2, 5, 7, 3, 4, 8 y 6.

Si se desean conocer las ventas de un artículo, hay que buscar el código del artículo recorriendo la columna 1. Si los datos de la columna 1 no están ordenados, el proceso de búsqueda del código del artículo es bastante dispendioso. Si los datos de la columna 1 están ordenados, el proceso de búsqueda será bastante fácil y eficiente.

El algoritmo que sigue ordena los datos de la matriz teniendo como base los datos de una columna dada:

```
1.  void ordenaPorCol(mat, m, n, c)        // mat: parámetro por referencia
2.      Variables: i, j, k:  numericaEntera
3.      INICIO
4.          PARA  i  DESDE  1  HASTA  m – 1   CON_VARIACION  +1
5.              k = i
6.              PARA  j  DESDE  i + 1  HASTA  m  CON_VARIACION  +1
7.                  SI  mat[j][c]  <  mat[k][c]
8.                      k = j
9.                  Fin(SI)
10.             Fin(PARA)
11.             intercambia_filas(mat, m, n, i, k)
12.         Fin(PARA)
13.     FIN
14. Fin(ordenaPorCol)
```

El método que utilizamos para efectuar el ordenamiento es el método de selección.

En la instrucción 1 definimos el subprograma con sus parámetros: **mat**, la matriz cuyos datos se desean ordenar; **m**, el número de filas de la matriz **mat**; **n**, el número de columnas de **mat**; y **c**, la columna que se toma como base para ordenar los datos de la matriz. Recuerde que el parámetro **mat** debe ser por referencia.

En la instrucción 2 definimos las variables de trabajo para ejecutar el proceso de ordenamiento: **i**, para identificar a partir de cuál fila faltan datos por ordenar; **j**, para determinar en cuál fila se halla el menor dato, de los datos de la columna **c;** y **k**, la variable en la cual almacenamos la fila que tiene el menor dato de la columna **c**.

Las instrucciones 4 a 10 de este algoritmo son similares a las instrucciones 4 a 10 del subprograma *ordenamientoAcendenteSelección(V, m)* del módulo 30. La única diferencia es que en la instrucción 7 comparamos el dato de la fila **j** columna **c**, con el dato de la fila **k** columna **c** para determinar en cuál fila se halla el menor dato de la columna **c**.

Al terminar el ciclo interno se ejecuta la instrucción 11, en la que se invoca el subprograma para intercambiar los datos de la fila **i** con los datos de la columna **k**, el cual fue desarrollado en el módulo anterior.

En la instrucción 12 se incrementa la **i** en 1 y continúa con el proceso de ordenamiento hasta que el contenido de la variable **i** sea igual al contenido de la variable **m**.

En la figura 37.1*b* se muestra cómo quedan los datos de la matriz después de ejecutar el proceso de ordenamiento.

Ordena por columna 1

(a)

	1	2	3	4	5	6	7	8 **n**
1	3	1	4	9	7	7	5	6
2	1	2	5	7	3	4	8	6
3	6	4	2	5	4	1	1	3
4	2	9	3	4	2	9	8	9
m 5	8	5	8	4	1	9	3	4

(b)

	1	2	3	4	5	6	7	8 **n**
1	1	2	5	7	3	4	8	6
2	2	9	3	4	2	9	8	9
3	3	1	4	9	7	7	5	6
4	6	4	2	5	4	1	1	3
m 5	8	5	8	4	1	9	3	4

Figura 37.1. Ordenamiento de los datos de una matriz con base en los datos de la columna 1.

Resumen

En este módulo hemos presentado un algoritmo, y su justificación, para ordenar los datos de una matriz con base en los datos de una columna.

Ejercicios propuestos

1. Una ensambladora de autos tiene un archivo con las **n** series de cada auto ensamblado, los costos de todas sus partes, los costos indirectos de fábrica y los costos de mano de obra. Haga un algoritmo para imprimir la serie y los datos de cada serie ordenados ascendentemente por costo real de fabricación, teniendo en cuenta que si el costo real de fabricación es inferior a 20 millones, el costo real que se debe reportar es de 20 millones. El costo real de fabricación debe determinarlo con una función que usted debe elaborar e invocar en el algoritmo principal.

2. Una universidad realizó inscripciones para sus diferentes programas académicos y generó un archivo con **m** registros, con el código del aspirante y el número de respuestas correctas (un entero entre 1 y 50) en cada una de las cuatro pruebas realizadas a los aspirantes. Sabiendo que el número del código de cada aspirante está dado por el orden en que se encuentran los registros en el archivo, haga un algoritmo para:

    - Imprimir el número de credencial y el promedio de respuestas correctas de cada aspirante.
    - Determinar e imprimir el número de aspirantes que ingresan a la universidad, sa-

biendo que sólo pasan los que hayan respondido correctamente mínimo el 60% de las preguntas.
- El número de credencial del aspirante con mejor promedio.
- El número de credencial del aspirante con peor promedio.

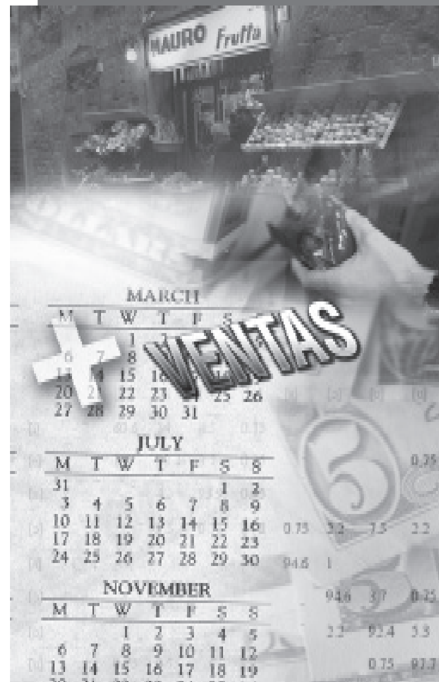
3. Un colegio tiene un archivo con el código y la fecha de nacimiento de cada uno de sus estudiantes. La fecha de nacimiento está descrita con tres campos: año, mes y día. Elabore un algoritmo que imprima en forma descendente por edad, el código del estudiante y su edad. Además, al final debe imprimir el promedio de edad de los estudiantes, el código del estudiante con mayor edad y el código del estudiante con menor edad. Recuerde que debe trabajar con subprogramas, los cuales también debe construir.

4. Dada una matriz con **m** filas y **n** columnas, en la cual cada celda contiene un dato numérico, elabore un algoritmo que ordene ascendentemente, por filas, todos los datos de la matriz, es decir, en la primera fila deben quedar los menores datos, en la segunda fila los siguientes y así sucesivamente.

5. Elabore un algoritmo que ordene descendentemente los datos de una matriz, con base en los datos de una fila.

Módulo 38

Construcción de la transpuesta de una matriz y suma de matrices



Introducción

Otras de las operaciones básicas con matrices son construir la transpuesta de una matriz y sumar dos matrices. Según la clase de matrices con las que se esté trabajando, los algoritmos con los que se ejecutan las diferentes tareas resultan diferentes. Más aún, dependiendo de la clase de matriz, habrá ciertas tareas que no están definidas para esa clase de matriz. En este módulo presentamos subprogramas correspondientes a construir la transpuesta de una matriz y a sumar dos matrices teniendo en cuenta las condiciones que deben cumplir para poder ejecutar dichas operaciones.

«Si utilizamos una matriz para representar las ventas de cada producto en los diferentes días del mes en cada una de las sucursales de una cadena de almacenes, estaremos interesados en conocer el consolidado de ventas en todas las sucursales. Para lograr obtener este consolidado debemos sumar las matrices de todos los almacenes».

Objetivos del módulo

1. Elaborar un algoritmo para construir la transpuesta de una matriz.
2. Elaborar un algoritmo para sumar dos matrices.

Preguntas básicas

1. ¿Qué es la transpuesta de una matriz?
2. ¿Cuál condición se debe cumplir para poder sumar dos matrices?
3. ¿En qué consiste sumar dos matrices?

Contenidos del módulo

38.1 Transpuesta de una matriz
38.2 Suma de dos matrices

Vea en el botón **Dos dimensiones** del mapa conceptual el video "Módulo 38. Operaciones con matrices: parte 2".

38.1 Transpuesta de una matriz

Dada una matriz **A** se define la matriz transpuesta de **A,** y la notaremos como $\mathbf{A}^t$, a la matriz que resulta de intercambiar las filas por las columnas de **A**, de tal forma que si **A** es una matriz de **m** filas y **n** columnas, su transpuesta resulta de **n** filas y **m** columnas.

Consideremos la figura 38.1 en la cual se presenta una matriz con su correspondiente transpuesta. Como se puede observar, la matriz original **A** tiene 5 filas y 7 columnas. La matriz $\mathbf{A}^t$ tiene 7 filas y 5 columnas.

Cada elemento de la fila **i** columna **j** de la matriz **A** queda ubicado en la fila **j** columna **i** de la matriz $\mathbf{A}^t$.

El dato que en la matriz **A** se hallaba en la fila 2 columna 5 queda en la fila 5 columna 2, y así sucesivamente.

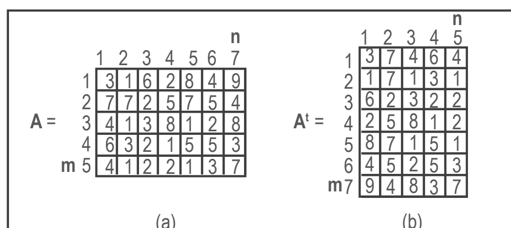


Figura 38.1. Matriz **A** con su transpuesta $\mathbf{A}^t$.

Un algoritmo con el cual se construye la transpuesta de una matriz se presenta a continuación:

```
1.   void transpuesta(A, m, n, At)   //  At: parámetro por referencia
2.           variables: i, j: numericaEntera
3.           INICIO
4.               PARA i DESDE 1 HASTA m CON_VARIACION +1
5.                   PARA j DESDE 1 HASTA n CON_VARIACION +1
6.                           At[j][i] = A[i][j]
7.                   Fin(PARA)
8.               Fin(PARA)
9.           FIN
10.  Fin(transpuesta)
```

En la instrucción 1 definimos el subprograma con sus parámetros: **A**, la matriz a la cual se desea construir la transpuesta; **m**, el número de filas de **A**; **n**, el número de columnas de **A**; y **At**, la variable en la cual quedará almacenada la transpuesta de **A**. El parámetro **At** debe ser por referencia.

En la instrucción 2 se definen las variables de trabajo **i** y **j**, con las cuales se recorre la matriz **A** y se construye **At**.

En las instrucciones 4 y 5 se plantean los ciclos para recorrer la matriz **A** por filas.

En la instrucción 6 se construye la transpuesta de **A**. El dato de la celda fila **i** columna **j** se le asigna a la matriz transpuesta en la fila **j** columna **i**.

Al terminar de ejecutar los ciclos de recorrido de **A**, por filas, en **At** queda la transpuesta de **A**.

38.2 Suma de dos matrices

La suma de dos matrices **A** y **B** consiste en crear una nueva matriz **C** en la cual cada elemento de **C** es la suma de los correspondientes elementos de las matrices **A** y **B**. Por lo tanto, para poder sumar dos matrices éstas tienen que tener las mismas dimensiones. Simbólicamente la suma de dos matrices se expresa así:

C[i][j] = A[i][j] + B[i][j]

El siguiente es un algoritmo con el cual se suman dos matrices:

```
1.  void suma(A, m, n, B, p, q, C)      // C: parámetro por referencia
2.      Variables: i, j: numericaEntera
3.      INICIO
4.          SI (m ≠ p || n ≠ q)
5.              ESCRIBA("las matrices no se pueden sumar")
6.              retorne
7.          Fin(SI)
8.              PARA i DESDE 1 HASTA m CON_VARIACION +1
9.                  PARA j DESDE 1 HASTA n CON_VARIACION +1
10.                     C[i][j] = A[i][j] + B[i][j]
11.                 Fin(PARA)
12.             Fin (PARA)
13.     FIN
14. Fin(suma)
```

En la instrucción 1 se define el subprograma con sus parámetros: **A** y **B,** las matrices que se van a sumar; **m** y **n,** las dimensiones de la matriz **A**; **p** y **q,** las dimensiones de la matriz **B**; y **C,** la matriz resultado. El parámetro **C** es un parámetro por referencia.

En la instrucción 2 definimos las variables de trabajo: **i** y **j**, para recorrer las matrices **A** y **B** por filas.

En la instrucción 4 se controla que las matrices **A** y **B** se puedan sumar. Para que dos matrices se puedan sumar deben tener las mismas dimensiones. Por lo tanto, si el número de filas de **A**, que es **m**, es diferente del número de filas de **B**, que es **p**, o el número de columnas de **A**, que es **n**, es diferente del número de columnas de **B**, que es **q**, las matrices no se pueden sumar y ejecutará las instrucciones 5 y 6 en las cuales produce el mensaje apropiado y retorna al programa llamante.

Si la condición de la instrucción 4 es falsa significa que las dos matrices **A** y **B** tienen las mismas dimensiones y por consiguiente continuará ejecutando la instrucción 8.

En las instrucciones 8 y 9 planteamos los ciclos para recorrer una matriz por filas.

En la instrucción 10 se le asigna a la celda de la fila **i** columna **j** de **C** el resultado de sumar el contenido de la celda fila **i** columna **j** de **A** con el contenido de la celda fila **i** columna **j** de **B**.

Al terminar de ejecutar los ciclos en la matriz **C** queda el resultado de sumar los datos de la matriz **A** con los datos de la matriz **B**.

En la figura 38.2 se muestra un ejemplo de suma de matrices.

Matriz **A**

	1	2	3	4	5	6 (n)
1	3	1	6	2	8	9
2	7	7	5	8	4	2
3	5	7	5	4	4	1
m 4	4	1	2	2	1	3

\+

Matriz **B**

	1	2	3	4	5	6 (n)
1	1	3	5	7	9	4
2	8	5	7	9	2	3
3	5	6	2	3	1	4
m 4	8	5	4	4	6	2

=

Matriz **C**

	1	2	3	4	5	6 (n)
1	4	4	11	9	17	13
2	15	12	12	17	6	5
3	10	13	7	7	5	5
m 4	12	6	6	6	7	5

Figura 38.2. Suma de matrices.

Observe que cada celda de la matriz **C** contiene la suma de sus correspondientes posiciones de las matrices **A** y **B**.

Resumen

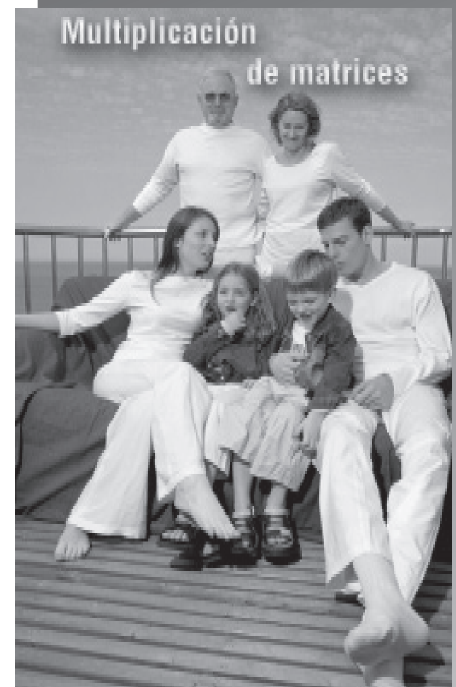
En este módulo hemos tratado dos de las operaciones básicas con matrices: construir la transpuesta de una matriz y sumar dos matrices. Para cada una de estas operaciones hemos mostrado un algoritmo que ejecuta dicha tarea.

Ejercicio propuesto

1. Elabore un algoritmo para sumar tres matrices, utilizando el subprograma de suma de dos matrices elaborado en el módulo.

Módulo 39

Multiplicación de matrices



Introducción

El producto de dos matrices es una operación que se efectúa con múltiples propósitos, dependiendo del problema que se esté trabajando. Para multiplicar dos matrices se deben cumplir ciertas condiciones. En este módulo presentamos las condiciones que se deben cumplir y un subprograma que efectúa dicho producto.

Objetivos del módulo

1. Elaborar un algoritmo que multiplique dos matrices.

Preguntas básicas

1. ¿Cuál condición se debe cumplir para multiplicar dos matrices?
2. Conocidas las dimensiones de las matrices que se van a multiplicar, ¿cuáles serán las dimensiones de la matriz resultado?

Contenidos del módulo

39.1 Multiplicación de matrices

«Una matriz es un objeto útil para representar relaciones. La relación «es madre de» se puede representar en una matriz; la relación «es esposa de» también se puede representar en una matriz. Teniendo representadas estas dos relaciones podremos construir la relación «es suegra de» efectuando el producto de la relación «es madre de» con la relación «es esposa de»».

Vea en el botón **Dos dimensiones** del mapa conceptual el video "Módulo 39. Multiplicación de matrices".

39.1 Multiplicación de matrices

Sea **A** una matriz de dimensiones **m * n,** siendo **m** el número de filas y **n** el número de columnas.

Sea **B** una matriz de dimensiones **p * q,** siendo **p** el número de filas y **q** el número de columnas.

Para poder efectuar el producto **A * B** de estas dos matrices se debe cumplir que **n == p**. Si **n** es diferente de **p** no se podrá efectuar el producto **A * B**.

La matriz resultante **C** tendrá dimensiones **m * q**, siendo **m** el número de filas de la matriz **A** y **q** el número de columnas de la matriz **B**.

Cada elemento **C[i][j]** de la matriz resultante es la sumatoria de los productos de los elementos **A[i][k] * B[k][j],** con **k** desde 1 hasta **n**.

Consideremos las matrices **A** y **B** de la figura 39.1. El producto **A * B** de ellas es la matriz **C**, que también se presenta en la misma figura.

En nuestro ejemplo, la matriz **A** tiene dimensiones **3 * 4**, es decir, **m** vale **3** y **n** vale **4**, y la matriz **B** tiene dimensiones **4 * 3**, o sea, **p** vale **4** y **q** vale **3**. Por consiguiente, el producto **A * B** se puede efectuar ya que **n** es igual a **p**, y las dimensiones de la matriz resultante son **3 * 3**, es decir, **3** filas y **3** columnas.

Simbólicamente, cada elemento de la matriz resultante **C** se describe así:

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} * B_{k,j}$$

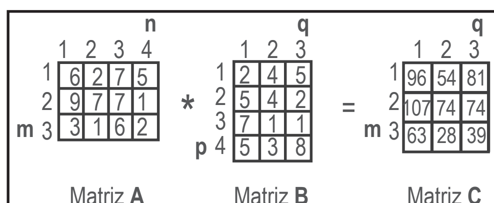


Figura 39.1. Multiplicación de dos matrices.

El siguiente subprograma controla y efectúa este proceso de multiplicación:

```
1.   void multMat(A, m, n, B, p, q, C)        // C: parámetro por referencia
2.        Variables: i, j, k:  numericaEntera
3.        INICIO
4.            SI  n ≠  p
5.                 ESCRIBA(“las matrices no se pueden multiplicar”)
6.                retorne
7.            Fin(SI)
8.                  PARA  i  DESDE  1  HASTA  m  CON_VARIACION  +1
9.                        PARA  j  DESDE  1  HASTA  q  CON_VARIACION  +1
10.                                C[i][j] = 0
11.                                PARA  k  DESDE  1  HASTA  n  CON_VARIACION  +1
12.                                       C[i][j] = C[i][j] + A[i][k] * B[k][j]
13.                                Fin(PARA)
14.                        Fin(PARA)
15.               Fin(PARA)
16.       FIN
17.   Fin(multMat)
```

En la instrucción 1 se define el subprograma con sus parámetros: **A** y **B,** las matrices a multiplicar; **m** y **n**, las dimensiones de la matriz **A**; **p** y **q,** las dimensiones de la matriz **B**; y **C**, la matriz resultante del producto. Recuerde que **C** debe ser un parámetro por referencia.

En la instrucción 2 definimos las variables de trabajo: **i**, para recorrer las filas de la matriz **A**; **j**, para recorrer las columnas de la matriz **B**; y **k**, para recorrer simultáneamente las celdas de una fila **i** de **A** y las celdas de una columna **j** de **B**.

En la instrucción 4 se controla que se pueda efectuar el producto **A * B**. Si el número de columnas de **A**, que es **n**, es diferente del número de filas de **B**, que es **p**, el producto no se podrá efectuar y, por lo tanto, ejecuta las instrucciones 5 y 6, con las cuales produce el mensaje apropiado y retorna al programa llamante.

Si la condición de la instrucción 4 es falsa, significa que el producto sí se puede ejecutar y, por lo tanto, no ejecuta las instrucciones 5 y 6 y continúa ejecutando la instrucción 8.

En la instrucción 8 se plantea el ciclo con el cual se recorrerá la matriz **A** por filas.

En la instrucción 9 se plantea el ciclo con el cual se recorrerá la matriz **B** por columnas.

En instrucción 10 se inicializa en cero el contenido de la celda fila **i** columna **j** de la matriz **C**. Esta inicialización es necesaria ya que dicha posición funciona como un acumulador.

En la instrucción 11 se plantea el ciclo con el cual se recorrerán las celdas de la fila **i** de **A** y las celdas de la columna **j** de **B**. Para un valor cualquiera de **k** (variable controladora del ciclo de la instrucción 11), se efectúa el producto del contenido de la celda fila **i** columna **k** de **A** con el contenido de la celda fila **k** columna **j** de **B** y se acumula este producto en la celda fila **i** columna **j** de **C**.

Al terminar el ciclo más interno (el ciclo de la instrucción 11), en la celda de la posición fila **i** columna **j** de **C** queda la sumatoria de los productos de los datos de las celdas de la fila **i** de **A** por los datos de las celdas de la columna **j** de **B**.

Resumen

En este módulo hemos tratado una operación básica con matrices: multiplicar dos matrices. Hemos mostrado la parte teórica correspondiente a este proceso y un algoritmo con el cual se efectúa dicha tarea.

Ejercicios propuestos

1. Haga seguimiento al algoritmo de multiplicación de matrices con los datos de la figura 39.1 y compruebe los resultados.

2. Elabore un subprograma que reciba como parámetros una matriz cuadrada y la dimensión de ella. Su algoritmo debe retornar verdadero si la matriz es un cuadrado mágico, y falso de lo contrario. Una matriz es un cuadrado mágico si la suma de los datos de cada fila, de cada columna y de cada diagonal es la misma.

3. Construya por programa las siguientes dos matrices

$$\mathbf{A} = \begin{bmatrix} 2 & 3 & 4 & \cdots & 08 \\ 3 & 4 & 5 & \cdots & 09 \\ \vdots & & & & \\ 08 & 09 & 10 & \cdots & 14 \end{bmatrix}_{7\times7} \qquad \mathbf{B} = \begin{bmatrix} 1 & 2 & 3 & \cdots & 7 \\ 2 & 3 & 4 & \cdots & 8 \\ \vdots & & & & \\ 7 & 8 & 9 & \cdots & 13 \end{bmatrix}_{7\times7}$$

   y calcule e imprima la matriz producto **A * B**. Luego calcule la matriz producto **B * A** e imprima **A * B** y **B * A**. Finalmente imprima un mensaje que diga si **A * B** es igual a **B * A**.

4. Dada una matriz **A** con **m** filas y **n** columnas, en la cual cada celda contiene un dato numérico entero, elabore un subprograma que retorne verdadero o falso dependiendo de si la matriz tiene punto de silla o no. Se dice que una matriz tiene punto de silla si existe un elemento **A[i][j]** tal que es el menor de la fila **i** y el mayor de la columna **j.** Tenga en cuenta que el menor de la fila **i** debe ser único, es decir, el menor dato sólo puede estar una vez en esa fila **i**; igualmente sucede con el mayor: también debe ser único.

5. Elabore un subprograma que reciba como parámetro una matriz cuadrada de orden 9. Su algoritmo debe retornar verdadero si los datos de la matriz conforman un sudoku, y falso de lo contrario. Los datos de una matriz conforma un sudoku si los datos de cada fila, de cada columna y de cada subregión de **3 * 3** son todos los dígitos del 1 al 9.

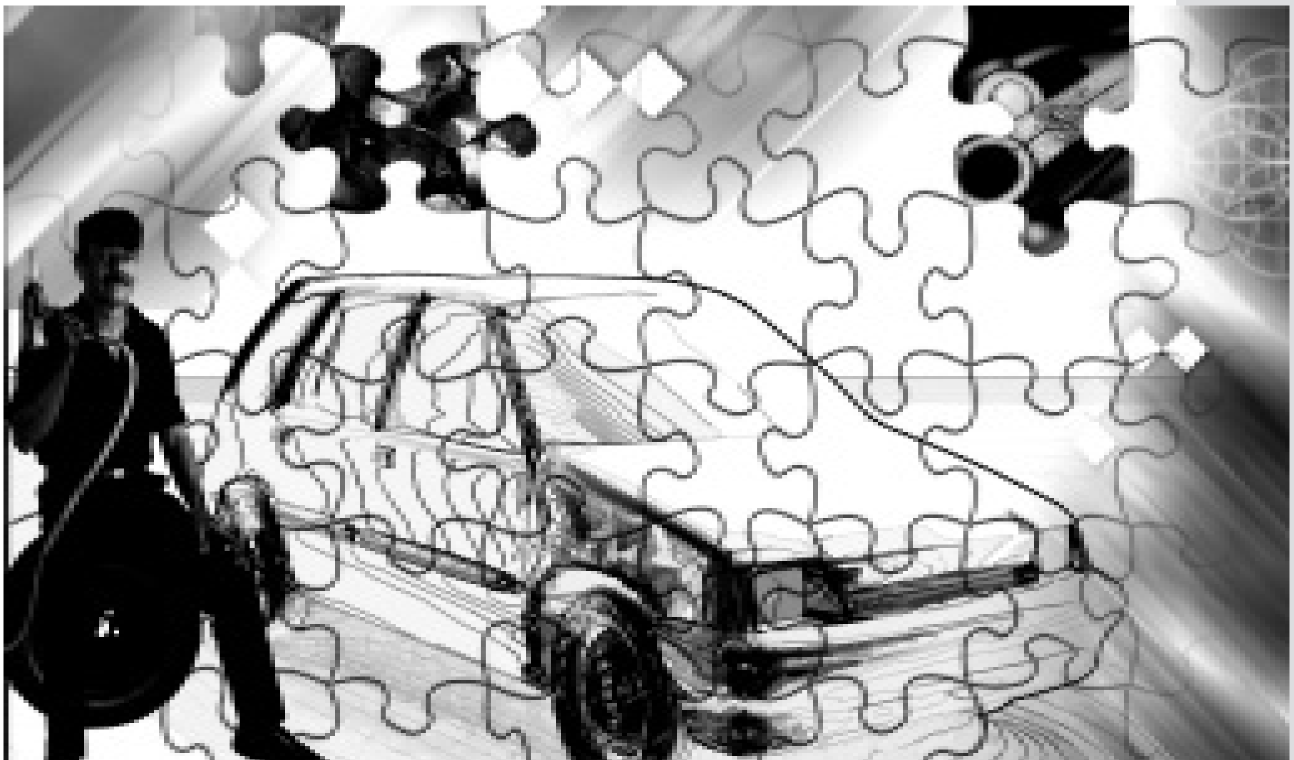
Capítulo 8 Programación orientada a objetos

Contenido breve

Módulo 40
Introducción a la programación orientada a objetos
Módulo 41
Desarrollo de los métodos de la clase vector
Módulo 42
Herencia y polimorfismo dinámico en vectores
Módulo 43
La clase matriz

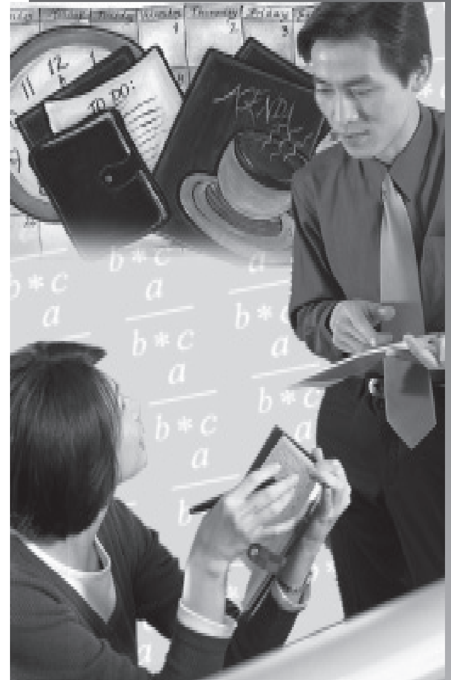
La programación orientada a objetos o POO (OOP, según sus siglas en inglés) es un paradigma de programación que usa objetos y sus interacciones para diseñar aplicaciones y programas informáticos. Está basada en varias técnicas, las cuales incluyen herencia, abstracción, polimorfismo y encapsulamiento. Su uso se popularizó a principios de la década de los noventa. En la actualidad hay gran variedad de lenguajes de programación que soportan la orientación a objetos.

La construcción de grandes obras siempre es el acople de pequeñas obras.



Módulo 40

Introducción a la programación orientada a objetos



Introducción

La programación orientada a objetos (POO) es realmente un estilo de programación cuyo principal objetivo es simular el mundo real. En la actualidad tiene mucho auge. Lenguajes de programación como JAVA están diseñados para trabajar bajo esta modalidad. En este módulo presentaremos una breve descripción acerca de cómo se desarrollan programas bajo este paradigma.

Objetivos del módulo

1. Conocer los principios básicos de la programación orientada a objetos.

Preguntas básicas

1. ¿Qué es programación orientada a objetos?
2. ¿Qué es una clase?
3. ¿Qué es polimorfismo?
4. ¿Qué es un atributo?
5. ¿Qué es un método?
6. ¿Qué es encapsulamiento?

Contenidos del módulo

40.1 Presentación
40.2 Definición de la clase vector
40.3 Prototipo de la clase vector

«La programación orientada a objetos es una metodología que descansa en el concepto de objeto para imponer la estructura modular de los programas. Permite comprender el dominio del problema a resolver, al intentar construir un modelo del mundo real que envuelve nuestro sistema. Es decir, la representación de este mundo mediante la identificación de los objetos que constituyen el vocabulario del dominio del problema, su organización y la representación de sus responsabilidades».

Vea en el botón **Objetual** del mapa conceptual el video "Programación orientada a objetos y clase vector".

40.1 Presentación

A lo largo del texto hemos venido trabajando con variables, cada una de las cuales se define de un *tipo* particular. El término *tipo* se refiere a la clase de datos que se pueden almacenar en una variable en algún lenguaje de programación. De esta manera se pueden definir variables de tipo entero, real, carácter, etc., que en nuestro seudocódigo hemos llamado numéricas enteras, numéricas reales, alfanumérica, etc.

Cuando se define una variable de tipo numérica entera significa que en esa variable sólo se podrán almacenar datos numéricos enteros. Sucede además que cuando definimos variables de tipo entero podemos efectuar una serie de operaciones con ellas (sumar, restar, multiplicar, dividir, etc.) y somos usuarios de dichas operaciones. Si definimos las variables **a, b** y **c** numericaEntera, cuando escribimos **c = a + b** la máquina o el compilador suma el contenido de **a** con el contenido de **b** y el resultado se lo asigna a **c**, y no nos preocupamos de cómo es el algoritmo de sumar. Es decir, las variables definidas de tipo numericaEntera tienen asociadas algunas operaciones por el hecho de ser de ese tipo. Sin embargo, cuando definimos variables que representan vectores o matrices no se dispone de esa cualidad. Para estructuras tales como vectores y matrices lo que se hizo fue que se construyeron subprogramas para las operaciones que se necesitan realizar con dichas estructuras, de tal manera que cuando se requiere efectuar alguna operación, con un vector por ejemplo, lo que hacemos es invocar el subprograma que efectúa ese proceso y le indicamos al subprograma con cuál vector debe trabajar. Esta forma de programar se conoce como *procedimental*. En este orden de ideas, lo que sucede es que los procesos priman sobre los datos.

En programación orientada a objetos (POO), las variables que representan estructuras como vectores o matrices se definirán asociadas a una clase.

Una clase es un tipo que tiene asociadas las operaciones que se pueden ejecutar con objetos de esa clase.

Viéndolo de esta forma, podremos decir que hablamos de la *clase* numericaEntera (en vez del *tipo* numericaEntera), es decir, un tipo que tiene asociadas ciertas operaciones, de las cuales sólo nos preocupamos por saber qué es lo que hacen, no cómo lo hacen.

Entonces, si definimos **a, b** y **c** de la clase numericaEntera, decimos que **a, b** y **c** son **objetos** de la clase numericaEntera.

Un **objeto** es una instancia de una **clase**

Cuando se define una clase, ésta por lo general consta de un conjunto de datos y un conjunto de operaciones que podrán efectuar objetos de esa clase. Los datos que pertenecen a una clase por lo general se definen *privados*, y se conocen como los *atributos* de esa clase. Las operaciones que pueden realizar los objetos de la clase son en realidad subprogramas, los cuales seguiremos llamando *métodos,* que pueden ser privados o públicos.

Para mostrar la diferencia principal entre programación procedimental y programación objetual consideremos el siguiente ejemplo. Sea Juan un objeto que pertenece a la clase serHumano. Consideremos además que existe un proceso denominado comer(persona, alimento). En programación procedimental, si Juan desea comer una pizza la forma de hacerlo es comer(Juan, pizza), lo cual significa que un programa principal (la mamá, por ejemplo) está invocando el proceso comer y le envía como parámetros a Juan y la pizza.

En POO la acción de comer es propia de cada objeto que pertenezca a la clase serHumano. Cada ser humano, por el mero hecho de pertenecer a la clase serHumano, tiene

incorporado ese proceso. Por lo tanto, si Juan desea comer pizza la forma de hacerlo es Juan.comer(pizza), es decir, el objeto Juan está ejecutando su propio proceso comer y recibe como parámetro lo que se va a comer: una pizza. En otras palabras, en POO los datos prevalecen sobre los procesos.

40.2 Definición de la clase vector

Vamos a presentar como primer ejemplo de cómo se define una clase, la definición de la clase vector.

Para la clase vector tendremos dos datos privados:

V: un arreglo de una dimensión (un vector).
n: el tamaño de dicho arreglo.

Vamos a considerar los arreglos de una dimensión de la manera como los definen los lenguajes de programación modernos tales como C# y JAVA. Es decir, cuando se defina un vector con **n** elementos los subíndices del vector toman los valores entre **0** y **n – 1**.

Si definimos **V[8]**: numericaEntera, o sea con **n = 8**, el vector **V** tendrá la siguiente forma:

	0	1	2	3	4	5	6	7
V								

El vector tiene 8 elementos, pero los datos (que siempre deben ser números enteros) se almacenan desde la posición 0 hasta la posición 7. Esto implica que los algoritmos que hemos desarrollado en el capítulo 6 para manipular vectores deben ser reformados para tener en cuenta esta característica.

Con el fin de obviar este inconveniente lo que haremos será definir el vector **V** con tamaño **n + 1**, y en la posición 0 de **V** tendremos el número de posiciones usadas en **V**. Así por ejemplo, cuando queramos definir un objeto de la clase vector, con capacidad para 8 elementos (**n** = 8), definiremos **V** de tamaño 9 (**n + 1**). La configuración inicial de **V** será:

	0	1	2	3	4	5	6	7	8
V	0								

la cual significa que el vector tiene tamaño 9, las posiciones en las cuales almaceno datos son de la 1 a la 8 y en el momento tengo 0 datos (**V[0] = 0**) almacenados en el vector. Y cuando dentro del proceso hayamos entrado datos al vector, podríamos estar en la siguiente situación:

	0	1	2	3	4	5	6	7	8
V	3	1	6	2					

la cual significa que el vector tiene tamaño 9, los datos se almacenan desde la posición 1 a la 8 y en el momento tengo almacenados 3 datos en el vector (**V[0] = 3**).

Teniendo esto definido veamos cómo definir la clase vector:

```
clase  vector
        Privado:
                n: numericaEntera
                V[]: objeto
Fin(vector)
```

Veamos lo que significa lo que hemos definido. Cuando se defina un objeto de la clase vector, cada objeto tendrá un arreglo **V** y éste tendrá tamaño **n + 1**.

Para definir un objeto de la clase vector se siguen los siguientes pasos:

1. Se define la variable igual que como se definen todas las variables a utilizar; simplemente, en vez de definirla de tipo numérica entera, numérica real, etc., se define de tipo vector. Por ejemplo, si tenemos la siguiente definición:

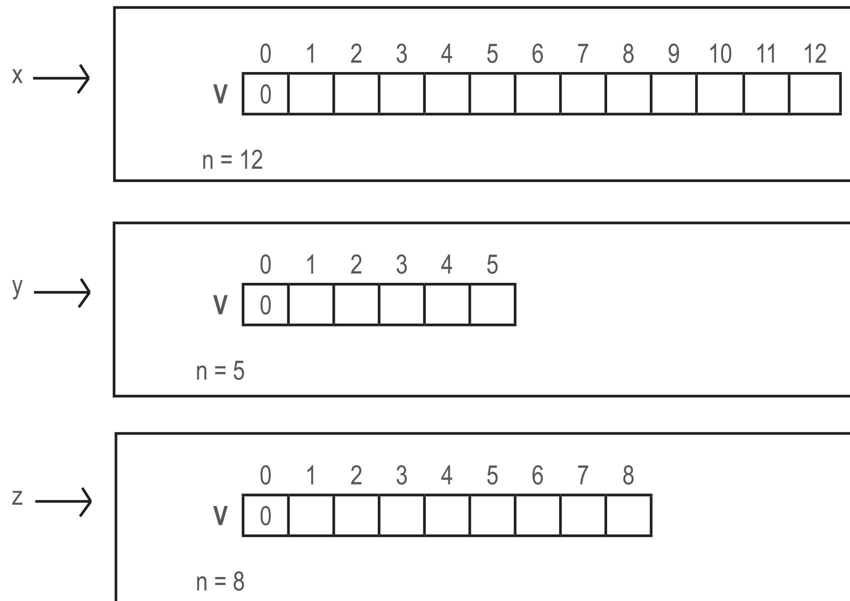
   ```
   Variables:  a, b, c: numericaEntera
               x, y, z: vector
   ```

   se están definiendo las variables **a, b** y **c** de tipo numérico entero, y las variables **x, y** y **z** de la clase vector.

2. Se crea la instancia de la clase. Para crear la instancia se usa el constructor. Para hacerlo utilizaremos la palabra clave **nuevo**. Por ejemplo, si deseamos crear las instancias correspondientes a las variables **x, y** y **z** que declaramos en el paso 1, escribimos las siguientes instrucciones:

   ```
   1.  x = nuevo  vector(12)
   2.  y = nuevo  vector(5)
   3.  z = nuevo  vector(8)
   ```

   las cuales al ejecutarse producen el siguiente efecto:



Fíjese que cada objeto tiene sus propios datos privados **V** y **n**.

Continuemos mostrando cómo se define la clase vector incluyendo el constructor:

```
Clase  vector
        Privado:
                n: numericaEntera
                V[]: objeto
        Público:
                vector(numericaEntera  nd)
Fin(vector)
```

El constructor es un subprograma público, que no es tipo void ni retorna ningún dato. Simplemente es el constructor. El constructor siempre tiene el mismo nombre de la clase. El constructor puede tener parámetros o no. En nuestra clase vector el constructor tiene un parámetro **nd**, el cual indica el tamaño del arreglo **V** que tendrá el objeto que se define. Para nuestra clase vector el constructor es:

```
1.  vector (numericaEntera  ne)
2.    INICIO
3.            n = ne
4.            V = new  objeto[n + 1]
5.            V[0] = 0
6.    FIN
7.  Fin(vector)
```

Fíjese que en nuestro constructor se está inicializando el dato privado **n** con el dato **ne** enviado como parámetro. Además se está creando el arreglo **V** con **n + 1** elementos, cada uno de los cuales tiene la capacidad de almacenar cualquier dato (se está definiendo **objeto** en la instrucción 3) y la posición 0 de **V** se inicializa en 0 indicando que el arreglo **V** está vacío.

Es conveniente hacer notar lo que significa la palabra objeto en la definición de **V** con la instrucción 3. La definición de **V** como objeto quiere decir que en las posiciones del vector se podrá almacenar cualquier tipo de dato: numérica entera, numérica real, alfanumérica, etc. Recuerde que en el capítulo 6, cuando se definía un vector, se especificaba qué tipo de datos se podían almacenar en el vector. Si se definía el vector de tipo numericaEntera significaba que en ese vector sólo se pueden almacenar datos numéricos enteros y no más. Al definir el vector de tipo **objeto** se permite que en las posiciones del vector se puede almacenar cualquier tipo de dato. Esto proporciona una gran versatilidad a la clase vector. Lo importante aquí es que cuando se vaya a acceder el dato de una posición hay que hacerlo con un proceso denominado *casting*. Este proceso consiste simplemente en convertir un tipo en otro. En este texto sólo haremos lo elemental de este tema. En un curso de JAVA podrá ampliar este tema y ver lo correspondiente a clases genéricas. En lo que respecta a nuestro texto, la forma de hacer casting será preceder la instrucción de acceso con el tipo al cual se desea hacer la conversión entre paréntesis. Por ejemplo, si queremos acceder el dato de la posición **i** de **V**, y llevarlo a una variable de tipo entero, lo haremos así: **a = (numericaEntera)V[i]**. El dato que está en **V[i]**, que es un objeto, quedará como un entero.

Ya hemos definido los datos privados y el constructor. Pasemos a definir las operaciones que podrán realizar los objetos de la clase vector. Presentaremos cuáles son las operaciones que se necesitan efectuar. Luego, para cada operación definiremos un método. Recuerde que en el contexto de POO dichas operaciones se definen como métodos.

1. Un método que retorne la capacidad del arreglo **V**: retorneCapacidad() retorna el valor del dato privado **n**.

2. Un método que retorne el número de elementos en el vector: retorneNumeroElementos() retorna el número de posiciones utilizadas, es decir, retorna **V[0]**.

3. Un método para actualizar el número de elementos del vector: asignaNumeroElementos(m) actualiza el contenido de **V[0]**.

4. Un método que retorne el dato correspondiente a una posición dada: retornaDatoEn(i) retorna el dato que se halla en la posición **i** de **V**.

5. Un método que actualice el dato correspondiente a una posición dada: asignaDato(d, i) almacena el dato **d** en la posición **i** de **V**.

6. Un método para imprimir los datos del vector: muestraDatos() recorre **V** mostrando el contenido de cada posición.

7. Un método para determinar si el vector está vacío o no: esVacio() retorna verdadero si **V[0]** es cero, falso de lo contrario.

8. Un método para determinar si el vector está lleno o no: esLleno() retorna verdadero si **V[0]** es igual a **n**, falso de lo contrario.

9. Un método que retorne la posición en la cual se halla un dato en el vector: retornePosicionDato(d) retorna la posición en la cual se halla el dato **d** en **V**. Si no encuentra el dato retorna **V[0] + 1**.

10. Un método para ordenar los datos del vector en forma ascendente: ordenaAscendente() ordena los datos de **V** en forma ascendente.

11. Un método para ordenar los datos del vector en forma descendente: ordenaDescendente() ordena los datos de **V** en forma descendente.

12. Un método para insertar un dato al final del vector: agregar(d) inserta el dato **d** a continuación del último dato de **V**.

13. Un método para buscar dónde insertar un dato en un vector ordenado ascendentemente: buscaDondeInsertar(d) retorna la posición en la cual debe quedar el dato **d** para que los datos de **V** continúen ordenados en forma ascendente.

14. Un método para insertar un dato en una posición específica: inserta(d, i) inserta el dato **d** en la posición **i** de **V**. Recuerde que esto implica mover los datos de **V**.

15. Un método para insertar un dato en un vector con los datos ordenados ascendentemente: inserta(d) inserta el dato **d** en un vector que tiene los datos ordenados ascendentemente conservando dicho orden.

16. Un método para borrar un dato que está en una posición dada del arreglo **V**: borraDatoEn(i) elimina el dato que está en la posición **i** de **V**.

40.3 Prototipo de la clase vector

Nuestra clase vector con las operaciones definidas es:

```
1.    clase  vector
2.           Privado:
3.                  n: numericaEntera
4.                  V[]:  objeto
5.           Público:
6.                  vector(numericaEntera  ne)
7.                  numericaEntera  retorneCapacidad()
8.                  numericaEntera  retorneNumeroElementos()
9.                  void  asignaNumeroElementos(ne)
10.                 objeto  retornaDatoEn(i)
11.                 void  asignaDato(d, i)
12.                 void  muestraDatos()
13.                 lógico  esVacio()
14.                 lógico  esLleno()
15.                 numericaEntera  retornePosicionDato(d)
16.                 void  ordenaAscendente()
17.                 void  ordenaDescendente()
18.                 void  agregar(d)
19.                 numericaEntera  buscaDondeInsertar(d)
20.                 void  inserta(d, i)
21.                 void  inserta(d)
22.                 void  borraDatoEn(i)
23.   Fin(vector)
```

Los métodos definidos en las líneas 20 y 21 tienen el mismo nombre, pero el número de parámetros es diferente. Esto hace que el computador sea capaz de diferenciar cuándo utiliza uno y cuándo utiliza el otro. Si se invoca el método inserta con dos parámetros, ejecutará el método inserta de la línea 20, pero si se invoca el método inserta con un parámetro, ejecutará el método inserta de la línea 21. Obviamente, las tareas ejecutadas por ambos métodos son diferentes. Esta característica de poder definir más de un método con el mismo nombre es lo que se conoce como polimorfismo.

Fíjese también que para acceder los datos privados de un objeto sólo se puede hacer a través de métodos. En nuestro ejemplo dichos métodos son los definidos en las líneas 7, 8, 10 y 12. Esta restricción se denomina encapsulamiento.

Teniendo definida dicha clase pasemos a elaborar un programa en el cual seamos usuarios de ella.

```
1.    void  usoClaseVector()
2.           Variables: a, b:  vector
3.                   n, d, i:  numericaEntera
4.           INICIO
5.                  ESCRIBA("entre tamaño del vector a")
6.                  LEA(n)
7.                  a = nuevo  vector(n)
8.                  ESCRIBA("entre un dato, para terminar entre un 0")
9.                  LEA(d)
10.                 MIENTRAS   d  !=  0
11.                        SI  a.esLleno()
12.                            d = 0
```

```
13.                     DE_LO_CONTRARIO
14.                             a.agregar(d)
15.                             ESCRIBA("entre un dato, para terminar
                                        entre un 0")
16.                             LEA(d)
17.                     Fin(SI)
18.             Fin(MIENTRAS)
19.             a.muestraDatos()
20.             a.ordenaAscendente()
21.             a.muestraDatos()
22.             SI  !a.esLleno()
23.                     ESCRIBA("entre dato a insertar")
24.                     LEA(d)
25.                     i = a.buscaDondeInsertar(d)
26.                     a.inserta(d, i)
27.                     a.muestraDatos()
28.             Fin(SI)
29.             ESCRIBA("entre tamaño del vector b")
30.             LEA(n)
31.             b = nuevo  vector(n)
32.             ESCRIBA("entre un dato, para terminar entre un 0")
33.             LEA(d)
34.             MIENTRAS   d  !=  0
35.                     SI  b.esLleno()
36.                             d = 0
37.                     DE_LO_CONTRARIO
38.                             b.inserta(d, 1)
39.                             ESCRIBA("entre un dato, para terminar
                                        entre un 0")
40.                             LEA(d)
41.                     Fin(SI)
42.             Fin(MIENTRAS)
43.             b.muestraDatos()
44.             b.ordenaAscendentemente()
45.             SI  !b.esLleno()
46.                     ESCRIBA("entre dato a insertar")
47.                     LEA(d)
48.                     b.inserta(d)
49.                     b.muestraDatos()
50.             Fin(SI)
51.         FIN
52.  Fin(usoClaseVector)
```

En la instrucción 2 se definen dos objetos de la clase vector: **a** y **b**. Con la instrucción 7 se crea la instancia del objeto **a** siendo el valor leído en la instrucción 6 el tamaño del vector con el que se va a trabajar. En las instrucciones 8 a 18 se llevan datos al vector hasta llenarlo o hasta que el usuario no quiera entrar más datos. Para llevar datos al objeto **a** se usa el método agregar, por consiguiente los datos leídos se insertan al final del arreglo **V** correspondiente al objeto **a**. Observe bien la forma como el objeto a invoca los métodos que desea ejecutar. Con la instrucción 19 se escriben los datos del objeto **a**. Con la instrucción 20 se ordenan ascendentemente los datos del objeto **a**. Con la instrucción 21 se muestran nuevamente los datos del objeto **a**. Con la instrucción 22 se controla que el objeto **a** no esté lleno. De no estarlo, se lee un dato (instrucción 24), se busca dónde insertarlo (instrucción 25), se inserta (instrucción 26) y se imprimen

nuevamente los datos del objeto **a**. En esta ocasión hemos usado el método inserta que tiene dos parámetros.

Con las instrucciones 29 a 44 construimos el objeto **b** insertando datos al principio (instrucción 38) y luego lo imprimimos. Luego de tenerlo ordenado se procede a insertarle un nuevo dato con la instrucción 48, en la cual se invoca el método inserta que sólo tiene un parámetro. Realmente, el método inserta que sólo tiene un parámetro es equivalente a ejecutar el método que busca dónde insertar seguido de la ejecución del método inserta que tiene dos parámetros. Es decir, las instrucciones

```
i = a.buscaDondeInsertar(d)
a.insertar(d, i)
```

son equivalentes a

```
a.insertar(d)
```

Por último, es conveniente resaltar que la forma como un objeto invoca un método es:

NombreDelObjeto.metodo(parametros)

Resumen

En este módulo hemos presentado los conceptos básicos de la programación orientada a objetos y un ejemplo que ilustra dichos principios básicos. Además hemos presentado el prototipo completo de la clase vector.

Ejercicios propuestos

1. Responda las preguntas hechas al principio del módulo en el ítem “preguntas básicas”.

2. ¿Cuál es la diferencia principal entre el paradigma procedimental y el paradigma objetual?

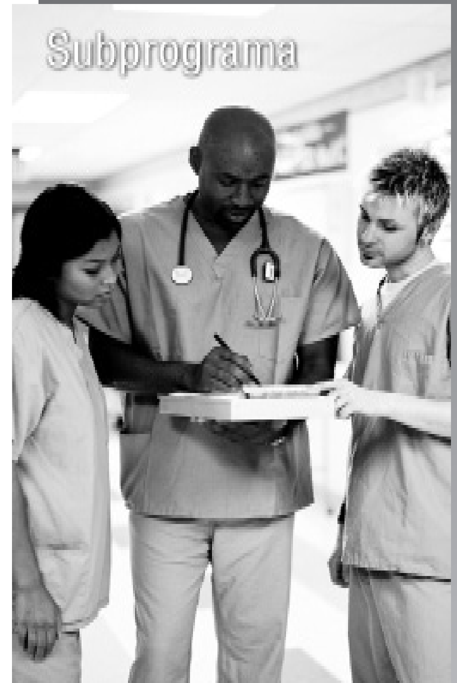
3. Construya un método, perteneciente a la clase vector, que elimine todas las ocurrencias de un dato **d** en un objeto de la clase vector, usando los métodos definidos para la clase vector.

4. Construya un algoritmo no perteneciente a la clase vector, que elimine todas las ocurrencias de un dato **d** en un objeto de la clase vector, usando los métodos definidos para la clase vector.

5. Escriba un algoritmo que construya un objeto de la clase vector leyendo un conjunto de datos, e insertando cada dato leído siempre al principio.

Módulo 41

Desarrollo de los métodos de la clase vector



Introducción

En el módulo anterior se definieron los principios básicos de la programación orientada a objetos y se presentó como ejemplo la clase vector, de la cual se elaboró su prototipo. Veremos aquí los algoritmos correspondientes a cada uno de los métodos definidos.

Objetivos del módulo

1. Elaborar los algoritmos correspondientes a los métodos definidos en el módulo 40.

Preguntas básicas

1. ¿Qué hace el método *retorneCapacidad()*?
2. ¿Qué hace el método *retorneNumeroElementos()*?
3. ¿Qué hace el método *asignaNumeroElementos(n)*?
4. ¿Qué hace el método *asignaDato(d, i)*?
5. ¿Qué hace el método *retornaDatoEn(i)*?
6. ¿Qué hace el método *esVacio()*?
7. ¿Qué hace el método *esLleno()*?
8. ¿Qué hace el método *muestraDatos()*?
9. ¿Qué hace el método *retornePosicionDato(d)*?
10. ¿Qué hace el método *agregar(d)*?
11. ¿Qué hace el método *intercambiar(i, j)*?
12. ¿Qué hace el método *ordenaAscendente()*?
13. ¿Qué hace el método *buscaDondeInsertar(d)*?
14. ¿Qué hace el método *inserta(d, i)*?
15. ¿Qué hace el método *borraDatoEn(i)*?

Contenidos del módulo

41.1 Desarrollo de los métodos
41.2 Ejemplo de uso

«El objetivo de POO es catalogar y diferenciar el código con base en estructuras jerárquicas dependientes, al estilo de un árbol. Los objetos se crean a partir de una serie de especificaciones o normas que definen cómo va a ser el objeto, lo cual se conoce como una clase. Las clases definen la estructura que van a tener los objetos que se creen a partir de ella, indicando los atributos y métodos que tendrán los objetos. Los atributos definen los datos o información del objeto, permitiendo modificar o consultar su estado, mientras que los métodos son las rutinas que definen el comportamiento del objeto. Es necesario tener muy clara cuál es la diferencia entre un objeto y una clase: una clase es la representación abstracta de algo, un objeto es una instancia de una clase».

41.1 Desarrollo de los métodos

Teniendo claro lo expuesto en el módulo anterior, nuestros algoritmos para los métodos definidos en el prototipo de la clase vector son exactamente los mismos que los desarrollados en el capítulo 6. La diferencia básica está en los parámetros. En los subprogramas elaborados en los módulos del capítulo 6 eran parámetros obligados el nombre del vector y el número de elementos en el vector. Aquí estos datos ya no se requieren como parámetros puesto que son atributos (datos) propios de cada objeto de la clase vector.

Vea en el botón **Objetual** del mapa conceptual el video "Algoritmos para métodos de vectores".

```
1.    numericaEntera  retorneCapacidad()
2.              INICIO
3.                    retorne(n)
4.              FIN
5.    Fin(retorneCapacidad)
```

```
1.    numericaEntera  retorneNumeroElementos()
2.              INICIO
3.                    retorne (V[0])
4.              FIN
5.    Fin(retornNumeroDeElementos)
```

```
1.    void  asignaNumeroElementos(ne)
2.              INICIO
3.                    V[0] = ne
4.              FIN
5.    Fin(asignaNumeroElementos)
```

```
1.    objeto  retornaDatoEn(i)
2.              INICIO
3.                    retorne(V[i])
4.              FIN
5.    Fin(retornaDatoEn)
```

```
1.    void  asignaDato(d, i)
2.              INICIO
3.                    V[i] = d
4.              FIN
5.    Fin(asignaDato)
```

Hasta aquí son métodos supremamente sencillos; basta una instrucción de asignación o una instrucción de retorno ya que lo único que se hace es acceder o modificar atributos privados de un objeto.

```
1.    void  muestraDatos()
2.         Variables: i: numericaEntera
3.          INICIO
4.              PARA  i  DESDE  1  HASTA  V[0]  CON_VARIACION  +1
5.                        ESCRIBA(i, V[i])
6.              Fin(PARA)
7.          FIN
8.    Fin(muestraDatos)
```

Este algoritmo es idéntico al del subprograma *imprimeVector(V, n)* del numeral 26.3. La diferencia está en que nuestro método ya no tiene parámetros puesto que el subprograma *muestraDatos()* lo invoca un objeto de la clase vector el cual sabe hasta dónde hay que recorrerlo, ya que **V[0]** contiene el número de elementos almacenados en **V**.

```
1.    lógico  esVacio()
2.          INICIO
3.              SI   V[0]  ==  0
4.                          retorne(verdadero)
5.              DE_LO_CONTRARIO
6.                          retorne(falso)
7.              Fin(SI)
8.          FIN
9.    Fin(esVacio)
```

Recuerde que **V[0]** contiene el número de datos almacenados en **V**. Por consiguiente, si en **V[0]** hay un 0 significa que el vector está vacío (retorna verdadero), de lo contrario hay datos en el vector entonces retorna falso. En lenguajes de programación como JAVA y C# las instrucciones 3 a 7 se abrevian así:

```
retorne(V[0] == 0)
```

Esta sola instrucción es equivalente a las instrucciones 3 a 7 que tenemos en nuestro método es*Vacio()*.

```
1.    lógico  esLleno()
2.          INICIO
3.              retorne (V[0] == n)
4.          FIN
5.    Fin(esLeno)
```

Aquí utilizamos la característica descrita en el método anterior.

```
1.    numericaEntera  retornePosicionDato(d)
2.          Variables:  i:  numeicaEntera
3.          INICIO
4.              i = 1
5.              MIENTRAS  i <= V[0]    &&    V[i]  !=  d
6.                          i = i + 1
7.              Fin(MIENTRAS)
8.              retorne(i)
9.          FIN
10.   Fin(retornePosicionDato)
```

Este algoritmo es idéntico al algoritmo *buscarPosicionDato(V, m, d)* elaborado en el numeral 29.1. Nuevamente la diferencia se presenta en los parámetros. El único parámetro requerido es el dato a buscar (el parámetro **d**), ya que nuestro método *retornePosicionDato(d)* es invocado por un objeto de la clase vector en el cual se sabe cuántos elementos se tienen almacenados, ya que **V[0]** mantiene esa información.

```
1.    void  ordenaAscendente()
2.          Variables: i, j, k:  numericaEntera
3.          INICIO
```

```
4.            PARA  i  DESDE  1  HASTA  V[0] – 1   CON_VARIACION  +1
5.                   k = i
6.                   PARA  j  DESDE  i + 1 HASTA  V[0]  CON_VARIACION  +1
7.                          SI  V[j] < V[k]
8.                                 k = j
9.                          Fin(SI)
10.                  Fin(PARA)
11.                  intercambiar(k, i)
12.           Fin(PARA)
13.       FIN
14.   Fin(ordenaAscendente)
```

Es el mismo algoritmo del numeral 30.2. La diferencia es que el método *ordenaAscendente()* no tiene parámetros (supongo que usted ya sabe por qué).

Aquí se usa un método que no hemos definido aún. En la instrucción 11 se invoca el método *intercambiar(k, i)*. Este es un método que se puede definir privado, y lo único que hace es intercambiar los datos de dos posiciones enviadas como parámetros. El método es:

```
1.    void  intercambiar(i, j)
2.          Variables: aux: objeto
3.          INICIO
4.                aux = V[i]
5.                V[i] = V[j]
6.                V[j] = aux
7.          FIN
8.    Fin(intercambiar)
```

Es muy importante hacer caer en la cuenta en este punto que el llamado al método *intercambiar(i, j)* en la instrucción 11 del algoritmo de ordenamiento ascendente no está precedido por ningún objeto. Esto significa que el objeto que invoca intercambiar es el mismo objeto que invocó *ordenaAscendente()*.

Este es un fenómeno bastante importante de entender. Cuando usted desarrolla un algoritmo para un método de una clase cualquiera, y dentro de ese algoritmo use otros métodos definidos para la misma clase, se puede obviar el objeto que invoca el método. Por defecto trabajará con el método que invocó el método que se está desarrollando.

```
1.    void  ordenaDescendente()
2.          Variables: i, j, k: numericaEntera
3.          INICIO
4.            PARA  i  DESDE  1  HASTA  V[0] – 1   CON_VARIACION  +1
5.                   k = i
6.                   PARA  j  DESDE  i + 1 HASTA  V[0]  CON_VARIACION  +1
7.                          SI  V[j] > V[k]
8.                                 k = j
9.                          Fin(SI)
10.                  Fin(PARA)
11.                  intercambiar(k, i)
12.           Fin(PARA)
13.       FIN
14.   Fin(ordenaDescendente)
```

Creo que no amerita ningún comentario. La diferencia con el algoritmo de *ordenaAscendente()* está en la instrucción de comparación de datos (instrucción 7). De todas formas, hay que tener cuidado con la instrucción 11: el objeto que invoca intercambiar es el objeto que invocó *ordenaDescendente()*.

```
1.    void  agregar(d)
2.          INICIO
3.                SI   esLleno()
4.                            ESCRIBA(“vector lleno, no se pueden agregar datos”)
5.                            retorne
6.                DE_LO_CONTRARIO
7.                            V[0] = V[0] + 1
8.                            V[V[0]] = d
9.                Fin(SI)
10.         FIN
11.   Fin(agregar)
```

Aquí, en la instrucción 3 se invoca el método *esLleno()*. ¿Quién lo invoca? El objeto que invocó *agregar(d).*

```
1.    numericaEntera  buscaDondeInsertar(d)
2.          Variables: i: numericaEntera
3.          INICIO
4.                i = 1
5.                MIENTRAS  i <= V[0]   &&   V[i] < d
6.                            i = i + 1
7.                Fin(MIENTRAS)
8.                retorne(i)
9.          FIN
10.   Fin(buscarDondeInsertar)
```

Este algoritmo es idéntico al del numeral 28.1. La diferencia son los parámetros.

```
1.    void  inserta(d, i)
2.          Variables: j: numericaEntera
3.          INICIO
4.                SI  esLleno()
5.                            ESCRIBA(“vector lleno, no se puede insertar”)
6.                            retorne
7.                Fin(SI)
8.                PARA  j  DESDE  V[0]  HASTA  i  CON_VARIACION   –1
9.                            V[j + 1] = V[j]
10                Fin(PARA)
11.               V[j] = d
12.               V[0] = V[0] + 1
13.         FIN
14.   Fin(inserta)
```

Este algoritmo es idéntico al del numeral 28.2. Nuevamente la diferencia son los parámetros.

```
1.    void  inserta(d)
2           Variables:  i:  numericaEntera.
```

```
3.        INICIO
4.            i = buscaDondeInsertar(d)
5.            inserta(d, i)
6.        FIN
7.    Fin(inserta)
```

Nuestro algoritmo *inserta(d)* es para facilitar la escritura de programas. El usuario simplemente invoca este subprograma y él se encarga de efectuar el proceso de búsqueda y de inserción. Recuerde que los métodos *buscaDondeInsertar(d)* e *inserta(d, i)* de las líneas 4 y 5 son invocados por el objeto que invocó el método *inserta(d)*.

```
1.    void  borraDatoEn(i)
2.        Variables: j: numericaEntera
3.        INICIO
4.            SI  i == V[0] + 1
5.                    ESCRIBA("dato no existe")
6.                    retorne
7.            Fin(SI)
8.            PARA  j  DESDE  i + 1  HASTA  V[0]  CON_VARIACION  +1
9.                    V[j – 1] = V[j]
10.           FIN(PARA)
11.           V[0] = V[0] – 1
12.       FIN
13.   Fin(borrar)
```

Este algoritmo es idéntico al del numeral 29.2.

41.2 Ejemplo de uso

Consideremos nuevamente el problema presentado en el módulo 25 referente a un censo en el Valle de Aburrá. Presentaremos una nueva solución usando objetos de la clase vector. Trabajaremos dos objetos de la clase vector: uno en el cual se almacenan los nombres de los municipios y otro en el cual tendremos el total de habitantes de cada municipio. Llamémoslos **nomMpio** y **acuMpio,** respectivamente. Tendremos dos archivos: uno en el que estarán los nombres de los municipios y otro en el que estarán los datos del censo. Primero procesaremos el archivo con los nombres de los municipios y luego el archivo con los datos del censo.

```
1.    void   programaCenso()
2.        Variables:  nomMpio, acuMpio:  vector
3.                    nombre, dir:  alfanumérica
4.                    mpio, np, i, temp:  numericaEntera
5.        INICIO
6.            nomMpio = nuevo vector(125)
7.            ABRA(nombreMpios)
8.            MIENTRAS  no  EOF(nombreMpios)
9.                    LEA(nombreMpios: nombre)
10.                   nomMpio.agregar(nombre)
11.           Fin(MIENTRAS)
12.           CIERRE(nombreMpios)
13.           nomMpio.ordenaAscendente()
14.           nomMpio.muestraDatos()
15.           acuMpio = nuevo vector(125)
16.           acuMpio.inicializaEnCeros()
```

```
17.             ABRA(censo)
18.             MIENTRAS  no EOF(censo)
19.                     LEA(censo: mpio, dir, np)
20.                     temp = (numericaEntera)acuMpio.retornaDatoEn(mpio) + np
21.                     acuMpio.asignaDato(temp, mpio)
22.             Fin(MIENTRAS)
23.             CIERRE(censo)
24.             PARA i   DESDE   1   HASTA   125   CON_VARIACION  +1
25.                     nombre = (alfanumerica)nomMpio.retornaDatoEn(i)
26.                     np = (numericaEntera)acuMpio.retornaDatoEn(i)
27.                     ESCRIBA(nombre, np)
28.             Fin(PARA)
29.             np = acuMpio.sumaDatos()
30.             ESCRIBA("el total de habitantes es: ", np)
31.         FIN
32.     Fin(programaCenso)
```

En las instrucciones 7 a 12 se crea el objeto **nomMpio**, el cual contiene los nombres de los municipios. Teniendo creado este objeto se procede a ordenarlo en forma ascendente y a imprimirlo con el fin de conocer el código asignado a cada municipio. Teniendo el código de los municipios se crea el archivo "**censo**", el cual se procesa en las instrucciones 17 a 23 creando el objeto **acuMpio** en el cual queda el total de personas de cada municipio. Luego, con las instrucciones 24 a 28 se imprime el nombre de cada municipio con sus habitantes y, por último, con la instrucción 29 se suman los datos del vector **acuMpio** y se imprime el total con la instrucción 30. Observe que en la instrucción 29 se invoca el método *sumaDatos()*, el cual, como su nombre lo dice, suma todos los datos de un objeto de la clase vector.

Resumen

En este módulo hemos presentado el desarrollo de los algoritmos correspondientes a los métodos definidos para la clase vector en el módulo anterior. Es importante observar la diferencia en cuanto a parámetros con respecto a los algoritmos presentados en los módulos 25 a 31.

Ejercicios propuestos

1. Construya el algoritmo correspondiente al método *inicializaEnCeros()* invocado en la instrucción 16 del algoritmo *programaCenso()* desarrollado en el numeral 41.2.

2. Construya el algoritmo correspondiente al método *sumaDatos()* invocado en la instrucción 29 del algoritmo *programaCenso()* desarrollado en el numeral 41.2.

3. Construya un método que retorne la posición en la cual se halla el mayor dato en un objeto de la clase vector.

4. Construya un método que retorne la posición en la cual se halla el menor dato en un objeto de la clase vector.

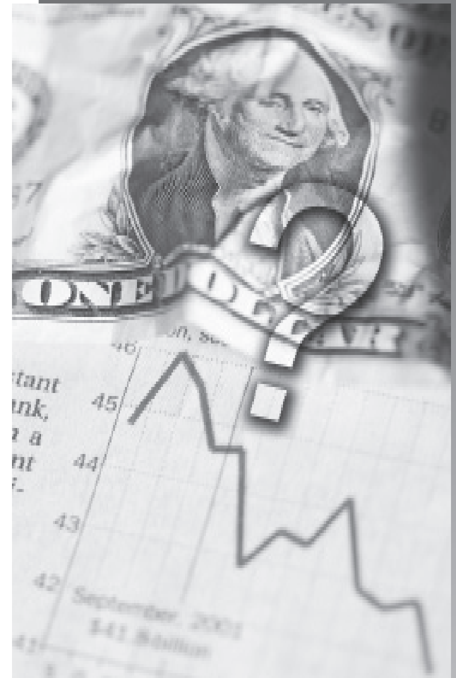
5. Se tienen dos objetos **a** y **b**, ambos de la clase vector, en los cuales los datos se hallan ordenados ascendentemente y tanto los datos en **a** como los datos en **b** son únicos,

pero puede haber datos comunes a ambos objetos. Elabore un método que construya un tercer objeto **c**, también de la clase vector, que sea la intercalación de los datos de los objetos **a** y **b**. En **c** no deben quedar datos repetidos. Los datos del objeto **c** deben quedar ordenados ascendentemente a medida que se va construyendo.

6. Se tienen dos objetos **a** y **b**, ambos de la clase vector. Elabore un método que construya un tercer objeto **c**, también de la clase vector, que sea la intersección de los datos que hay en **a** con los datos que hay en **b**.

Módulo 42

Herencia y polimorfismo dinámico en vectores



Introducción

En el ejercicio 3 del módulo 31 se propuso manejar números enteros de gran tamaño representados en vectores, de a dígito por posición. En la guía de autoevaluación se presentó una solución. Vamos a tratar aquí este ejercicio de una manera objetual. Espero que sea de gran ayuda para entender mejor las diferencias y las mejoras bajo el paradigma objetual.

Objetivo del módulo

1. Conocer la propiedad de herencia, sus conceptos y su aplicación.

Preguntas básicas

1. ¿Qué es herencia?
2. ¿Qué es una clase base?
3. ¿Qué es una clase derivada?
4. ¿Qué es una subclase?
5. ¿Qué cualidad tienen los atributos definidos protegidos?
6. ¿Qué es sobrecarga de operadores?
7. ¿Qué significa la palabra reservada this?
8. ¿Qué es polimorfismo dinámico?

Contenidos del módulo

42.1 Enunciado del ejercicio y representación
42.2 Desarrollo del ejercicio

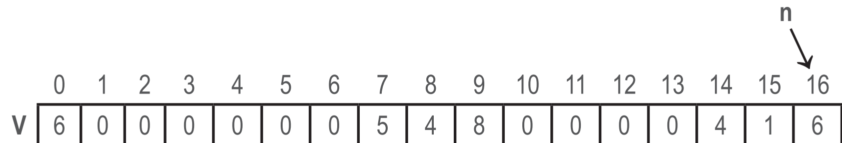
«Una clase puede heredar sus atributos y métodos a varias subclases (la clase que hereda es llamada superclase). Esto significa que una subclase, aparte de los atributos y métodos propios, tiene incorporados los atributos y métodos heredados de la superclase. Una subclase puede a su vez comportarse como una superclase y heredar a otras clases, creando de esta manera la jerarquía de herencia. Cuando una clase se hereda de más de una superclase se conoce como herencia múltiple».

Vea en el botón **Objetual** del mapa conceptual el video "Aplicación de vector por herencia y polimorfismo dinámico".

42.1 Enunciado del ejercicio y representación

Se tienen dos objetos **a** y **b**. En cada uno de ellos se halla almacenado un número entero, de a dígito por posición. Elabore subprogramas para sumar, restar, multiplicar y dividir números enteros representados en esa forma.

Recordemos la forma de representar enteros. Si tenemos el número 5480000416 y lo queremos representar en un vector de 16 elementos, el vector será:

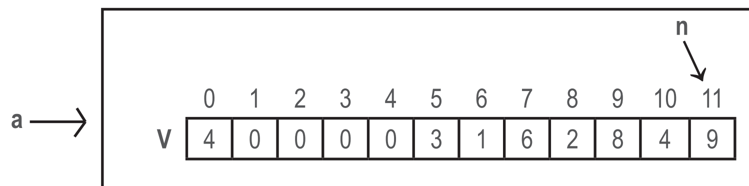


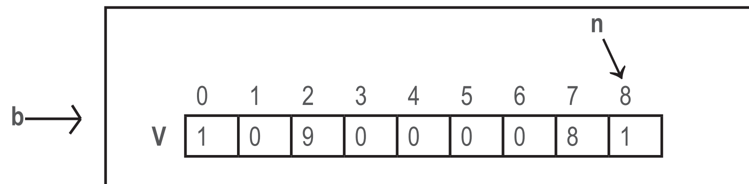
Fíjese que el dígito menos significativo se halla en la posición **n** del vector, y como el vector tiene una capacidad de 16 elementos, los elementos más hacia la izquierda son 0. Aquí introducimos una pequeña modificación con respecto al dato que se maneja en la posición 0 del vector. En los módulos anteriores en la posición 0 se maneja el número de posiciones ocupadas, pero aquí, en la posición 0 se tendrá la posición anterior a la cual comienza el número que se representa. En nuestro ejemplo, **V[0]** vale 6, lo cual significa que las posiciones desde la 1 hasta la 6 son 0. Es importante también darse cuenta de que el número de dígitos que tiene un número representado de esta forma es **n** menos el contenido de la posición 0 del arreglo **V**. Instruccionalmente:

nd = objeto.retorneCapacidad() – objeto.retorneNumeroElementos()

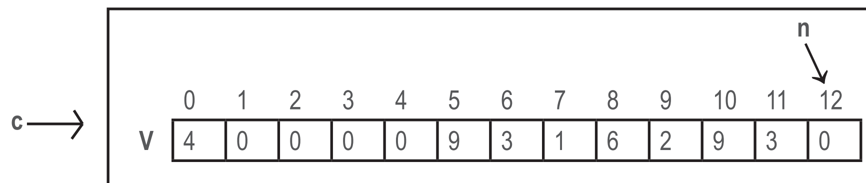
siendo **nd** número de dígitos.

Consideremos los números representados en los objetos **a** y **b** en los cuales se hallan almacenados dos números enteros:





Al ejecutar la suma de ellos el resultado quedará en el objeto **c** que mostramos a continuación:



Definiremos una clase altaPrecision, la cual será **derivada** de la clase vector. Esto implica que todos los métodos definidos para la clase vector podrán ser utilizados por objetos de la clase altaPrecision. Básicamente esto es lo que se conoce como reutilización de código. Para la clase altaPrecisión se redefine el constructor, ya que tiene una pequeña variación en **V[0]**, y se incluyen los métodos que son propios de la clase que se está definiendo: suma, resta, multiplica y divide.

```
1.    clase  altaPrecision derivada de vector
2.          Público:
3.                altaPrecision(numericaEntera  ne)          // constructor
4.                altaPrecision  suma(altaPrecision  b)
5.                altaPrecision  resta(altaPrecision  b)
6.                altaPrecision  multiplica(altaPrecision  b)
7.                altaPrecision  divide(altaPrecision  b)
8.                void  muestraDatos()
9.    Fin(altaPrecision)
```

Cuando se define una clase derivada de otra clase, esta última se denomina **clase base**, y la clase definida se llama **clase derivada** o **subclase**. Además, si se desea permitir que los objetos de las subclases puedan acceder los datos privados de la clase base, éstos deben definirse protegidos. Los atributos definidos **protegidos** podrán ser accedidos por los métodos de cualquier subclase.

Fíjese que los métodos definidos para las operaciones de suma, resta, multiplicación y división están definidos de altaPrecision, lo cual significa que retornan un objeto de la clase altaPrecision. Si queremos sumar dos números enteros **x** e **y** representados de esta forma entonces el objeto **x** invoca el método suma y le envía como parámetro el objeto **y**. El resultado de la ejecución es un nuevo objeto de la misma clase altaPrecision. Por ejemplo: z = x · suma(y).

El lenguaje de programación C# admite lo que se conoce como **sobrecarga de operadores,** lo cual permite construir instrucciones más claras. Si se sobrecarga el operador + en la definición de los métodos de nuestra clase altaPrecision podremos escribir: **z = x + y**. Como **x** e **y** son objetos de la clase altaPrecision, cuando ejecuta esta instrucción lo que sucede es que se ejecuta el método suma definido en la clase. El lenguaje JAVA no tiene esta propiedad.

Adicionalmente se define un método *muestraDatos()*, el cual imprime el número representado en un objeto perteneciente a la clase altaPrecision.

```
1.     void  muestraDatos()
2.        Variables:  i:  numericaEntera
3.        INICIO
4.              PARA   i   DESDE   V[0] + 1  HASTA   n   CON_VARIACION  +1
5.                    ESCRIBA(V[i])
6.              Fin(PARA)
7.        FIN
8.    Fin(muestraDatos)
```

Este método tiene el mismo nombre que un método definido en la clase base. Esta es otra forma de polimorfismo, conocida como *polimorfismo dinámico*. Cuando se tiene una clase derivada, los objetos de esta clase invocan métodos. Si encuentra el método en la clase derivada ejecuta ese método; si no, busca el método en la clase base y ejecuta el método de la clase base (reutilización). Si existen métodos que tienen el mismo nombre, los ob-

jetos pertenecientes a la clase derivada ejecutan el método definido en la clase derivada y los objetos pertenecientes a la clase base ejecutan el método definido en la clase base (polimorfismo dinámico).

Otro aspecto importante es el siguiente: en la instrucción 4 se accede el dato de la posición 0 del arreglo **V**. Esto se puede hacer si el atributo **V** fue definido como protegido en la clase base. Si el atributo **V** fue definido como privado no se podría acceder el dato de la posición 0 de esta forma, sino que habría que utilizar el método *retorneNumeroElementos()*.

42.2 Desarrollo del ejercicio

Veamos cómo quedan los algoritmos para el constructor, sumar, restar, multiplicar y dividir enteros con muchos dígitos representándolos en vectores bajo el paradigma objetual. Comencemos con el constructor:

```
1.    altaPrecision  (numericaEntera  ne)
2.          Variables:  i:  numericaEntera
3.          INICIO
4.                n = ne
5.                V = new  numericaEntera[n + 1]
6.                V[0] = ne
7.                PARA  i  DESDE  1   HASTA  n   CON_VARIACION  +1
8.                            V[i] = 0
9.                Fin(PARA)
10.         FIN
11.   Fin(altaPrecision)
```

Como podrá observar, el arreglo **V** se ha definido de tipo numérico entero puesto que en él sólo se almacenarán números enteros: un dígito en cada posición. Además, la posición 0 se ha inicializado en **n**, puesto que inicialmente todos los datos desde la posición 1 hasta la **n** son 0.

Continuemos con el método para sumar.

```
1.    altaPrecision  suma(altaPrecision  b)
2.          Variables: i, j, k, acarreo, q: numericaEntera
3.                      c:  altaPrecision
4.          INICIO
5.                m = retorneCapacidad()
6.                n = b.retorneCapacidad()
7.                p = mayor(m, n) + 1
8.                c = nuevo  altaPrecision(p)
9.                acarreo = 0
10.               i = m
11.               j = n
12.               k = p
13.               m = retorneNumeroElementos()
14.               n = b.retorneNumeroElementos()
15.               MIENTRAS  i > m   &&   j > n
16.                           q = retornaDatoEn(i) + b.retornaDatoEn(j) + acarreo
17.                           c.asignaDato(q % 10, k)
18.                           acarreo = q / 10
19.                           i = i – 1
20.                           j = j – 1
```

```
21.                     k = k – 1
22.             Fin(MIENTRAS)
23.             MIENTRAS  i > m
24.                     q = retornaDatoEn(i) + acarreo
25.                     c.asignaDato(q % 10, k)
26.                     acarreo = q / 10
27.                     i = i – 1
28.                     k = k – 1
29.             Fin(MIENTRAS)
30.             MIENTRAS   j > n
31.                     q = b.retornaDatoEn(j) + acarreo
32.                     c.asignaDato(q % 10, k)
33.                     acarreo = q / 10
34.                     j = j – 1
35.                     k = k – 1
36.             Fin(MIENTRAS)
37.             SI   acarreo  >  0
38.                     c.asignaDato(acarreo, k)
39.                     k = k – 1
40.             Fin(SI)
41.             c. actualizaNumeroDigitos()
42.             retorne(c)
43.         FIN
44.     Fin(suma)
```

Nuestro algoritmo tiene la misma estructura del algoritmo elaborado en el ejercicio 3 del módulo 31 de la guía de autoevaluación. Resaltemos las diferencias entre éste y el elaborado en dicha guía. En la instrucción 3 se ha definido una variable local, de tipo altaPrecisión, llamada **c,** en la cual se almacenará el resultado y se retornará. El subprograma *inicializaVector()* ya no se requiere puesto que el constructor efectúa esa tarea. Con las instrucciones 5 y 6 se obtiene el número de elementos del vector y con las instrucciones 13 y 14 se obtienen las posiciones hasta donde se procesan los vectores. Observe que en las instrucciones 5 y 13 se invocan los métodos *retorneCapacidad()* y *retorneNumeroElementos()* sin precederlos de ningún objeto, mientras que en las instrucciones 6 y 14 se invocan los mismos métodos precedidos por el objeto **b**. En las instrucciones 6 y 14 el objeto **b** es el que invoca dichos métodos. En las instrucciones 5 y 13 los métodos son invocados por el objeto que invocó el método suma. Este es un aspecto muy importante a tener en cuenta: cuando se está elaborando el algoritmo correspondiente a un método de una clase, y dentro de este algoritmo se invoca otro método (met) perteneciente a la misma clase y no se coloca el objeto que invoca el método met, significa que el método met está siendo invocado por el objeto que invocó el método cuyo algoritmo se está desarrollando. Con el fin de evitar esta "confusión" muchos programadores optan por utilizar la palabra reservada **this**, la cual indica que el objeto que invoca el método met es el objeto que invocó el método cuyo algoritmo se está elaborando. En las instrucciones 16, 24 y 31 se invoca el método *retornaDatoEn(subindice)* con el que se accede el dato que está en la posición subíndice del arreglo **V**, el cual es un atributo privado de los objetos pertenecientes a la clase altaPrecision. Con las instrucciones 17, 25, 32, 38 y 41 se invoca el método *asignaDato(dato, subindice)*, el cual simplemente almacena el dato enviado como parámetro en la posición subíndice del arreglo **V** perteneciente al objeto que invocó el método *asignaDato(d, i)*. Con la instrucción 42 se retorna el objeto construido, el cual contiene el resultado de sumar el número almacenado en el objeto que invoca el método con el número almacenado en el objeto enviado como parámetro. Es importante hacer notar la simplificación en el paso de parámetros con respecto al algoritmo desarrollado en el ejercicio 3 del módulo 31 de la guía de autoevaluación.

Continuemos con el algoritmo para restar dos números representados en vectores, tal como lo hemos definido.

```
1.      altaPrecision  resta(altaPrecision  b)
2.          Variables: i, j, k, m, n, p:  numericaEntera
3.                  c, aux:  altaPrecision
4.          INICIO
5.              m = retorneCapacidad()
6.              n = b.retorneCapacidad()
7.              i = m
8.              j = n
9.              p = m
10.             c = nuevo  altaPrecision(p)
11.             aux = copiaNumero()
12.             m = retorneNumeroElementos()
13.             n = b.retorneNumeroElementos()
14.             k = p
15.             MIENTRAS   i > m   &&   j > n
16.                     SI   b.retornaDatoEn(j) > aux.retornaDatoEn(i)
17.                             aux.actualiza(i)
18.                     Fin(SI)
19.                     c.asignaDato(aux.retornaDatoEn(i) – b.retornaDatoEn(j), k)
20.                     i = i – 1
21.                     j = j – 1
22.                     k = k – 1
23.             Fin(MIENTRAS)
24.             MIENTRAS  i > m
25.                     c.asignaDato(aux.retornaDatoEn(i), k)
26.                     i = i – 1
27.                     k = k – 1
28.             Fin(MIENTRAS)
29.             c.actualizaNumeroDigitos()
30.             retorne(c)
31.         FIN
32.     Fin(resta)
```

Se recorren los vectores simultáneamente desde el último dígito hasta la posición en la cual empieza el número representado (ciclo desde la instrucción 15 hasta la 23). Con la instrucción 16 se controla que el dígito del minuendo sea mayor o igual que el dígito del sustraendo. De no ser así se actualiza el minuendo para poder efectuar la resta (instrucción 17). Como este proceso modifica los datos del número entrado como minuendo (el objeto que invocó el método), se hace una copia de éste para poder efectuar la resta y no alterar el minuendo. Dicha copia se efectúa con el método *copiaNumero()*, el cual veremos más adelante. El método que modifica los datos del minuendo se ha llamado *actualiza(i)* y también lo veremos a continuación. Terminado el ciclo principal (ciclo desde la instrucción 15 hasta la 23) se continúa con el ciclo de la instrucción 24 a la 28 en el cual se acaban de trasladar los dígitos del minuendo hacia el resultado en caso de que el minuendo tenga más dígitos que el sustraendo.

```
1.      altaPrecision  copiaNumero()
2.          Variables: i, n: numericaEntera
3.                  c: altaPrecision
4.          INICIO
5.              n = retorneCapacidad()
```

```
6.              c = nuevo  altaPrecision(n)
7.              PARA  i  DESDE  0   HASTA  n  CON_VARIACION  +1
8.                      c.asignaDato(retornaDatoEn(i), i)
9.              Fin(PARA)
10.             retorne(c)
11.         FIN
12.     Fin(copiaNumero)
```

Este es un algoritmo bastante sencillo. Simplemente se crea un nuevo objeto de la clase altaPrecision y se trasladan todos los datos del objeto que invocó el método hacia el nuevo objeto.

```
1.      void  actualiza(i)
2.          Variables: q, aux: numericaEntera
3.          INICIO
4.              q = i – 1
5.              MIENTRAS   retornaDatoEn(q) == 0
6.                      asignaDato(9, q)
7.                      q = q – 1
8.              Fin(MIENTRAS)
9.              aux = retornaDatoEn(q) – 1
10.             asignaDato(aux, q)
11.             aux = retornaDatoEn(i) + 10
12.             asignaDato(aux, i)
13.         FIN
14.     Fin(actualiza)
```

Nuestro método *actualiza(i)* es idéntico al desarrollado en el ejercicio 3 del módulo 31. La diferencia es que aquí se trata de una manera objetual.

Veamos ahora un algoritmo para multiplicar dos números enteros representados en vector tal como lo hemos definido.

```
1.      altaPrecision  multiplica(altaPrecision  b)
2.          Variables:  i, j, k, m, p, q, r, s, acarreo, aux:  numericaEntera
3.                      c:  altaPrecision
4.          INICIO
5.              m = retorneCapacidad()
6.              n = b.retorneCapacidad()
7.              p = m + n +1
8.              c = nuevo altaPrecision(p)
9.              r = retorneNumeroElementos()
10.             s = b.retorneNumeroElementos()
11.             PARA  i  DESDE  m  HASTA  r + 1  CON_VARIACION  –1
12.                 acarreo = 0
13.                 k = p – (m – i)
14.                 PARA  j  DESDE  n  HASTA  s + 1  CON_VARIACION  –1
15.                     aux = c.retornaDatoEn(k)
16.                     q = aux + retornaDatoEn(i) * b.retornaDatoEn(j) + acarreo
17.                     c.asignaDato(q % 10, k)
18.                     acarreo = q / 10
19.                     k = k – 1
20.                 Fin(PARA)
```

```
21.                 c.asignaDato(acarreo, k)
22.             Fin(PARA)
23.             c.asignaDato(acarreo, k)
24.             c.actualizaNumeroDigitos()
25.             retorne(c)
26.         FIN
27.     Fin(multiplica)
```

Este algoritmo es casi igual al desarrollado en el ejercicio 3 del módulo 31. Tiene una diferencia importante con respecto al mencionado. En el algoritmo desarrollado anteriormente se recorrían todas las posiciones del vector, incluyendo las posiciones que representan ceros a la izquierda. Aquí no se recorren estas posiciones, por lo cual se debe agregar la instrucción 21 con el fin de que no se pierda un acarreo que posiblemente puede ser diferente de cero. Se recomienda hacer prueba de escritorio a este algoritmo para que el estudiante lo entienda mejor.

Continuemos con el algoritmo para la división.

```
1.  altaPrecision  divide(altaPrecision  b)
2.      Variables:  x, m:  numericaEntera
3.                  c, d, aux: altaPrecision
4.      INICIO
5.          m = retorneCapacidad()
6.          c = nuevo altaPrecision(m)
7.          aux = nuevo altaPrecision(3)
8.          aux.asignaDato(1, 3)
9.          aux.asignaDato(2, 0)
10.         d = copiaNumero()
11.         x = d.compare(b)
12.         MIENTRAS  x  !=  – 1
13.                 c = c.suma(aux)
14.                 d = d.resta(b)
15.                 x = d.compare(b)
16.         Fin(MIENTRAS)
17.         c. actualizaNumeroDigitos()
18.         retorne(c)
19.     FIN
20. Fin(divide)
```

Este algoritmo es idéntico al desarrollado en el ejercicio 3 del módulo 31, pero con tratamiento objetual. Aprópiese bien de la forma como se simplifica este algoritmo al tener los números representados como objetos. Es mucho más legible la forma como se divide (ciclo de instrucciones 11 a 15): se le suma 1 al cociente (instrucción 12), se le resta el divisor al dividendo (instrucción 13) y se compara nuevamente el dividendo con el divisor (instrucción 14) para controlar el fin de la división.

El algoritmo para comparar dos números representados bajo esta forma difiere del elaborado en el ejercicio 3 del módulo 31. En aquella solución considerábamos los vectores del mismo tamaño. Aquí consideraremos el caso más general, en el cual los vectores pueden ser de diferente tamaño.

```
1.  numericaEntera  compare(altaPrecision  b)
2.      Variables: i, j, m, n:  numericaEntera
3.      INICIO
```

```
4.          m = retorneCapacidad() – retornaDatoEn(0)
5.          n = b.retorneCapacidad() – b.retornaDatoEn(0)
6.          SI  m > n
7.                  retorne(+1)
8.          Fin(SI)
9.          SI  m < n
10.                 retorne(– 1)
11.         Fin(SI)
12.         i = retornaDatoEn(0) + 1
13.         j = b.retornaDatoEn(0) + 1
14.         m = retorneCapacidad()
15.         MIENTRAS   i <=  m   &&    retornaDatoEn(i)  ==  b.retornaDatoEn(j)
16.                 i = i + 1
17.                 j = j + 1
18.         Fin(MIENTRAS)
19.         SI  i > m
20.                 retorne(0)
21.         Fin(SI)
22.         SI   retornaDatoEn(i)  >  b.retornaDatoEn(j)
23.                 retorne(+1)
24.         DE_LO_CONTRARIO
25.                 retorne(– 1)
26.         Fin(SI)
27.     FIN
28. Fin(compare)
```

Con las instrucciones 4 y 5 se determina el número de dígitos de cada número. Si el número que invocó el método tiene más dígitos que el número enviado como parámetro retorna +1 (instrucciones 6 a 8). Si el número que invocó el método tiene menos dígitos que el número enviado como parámetro retorna –1 (instrucciones 9 a 11). En caso de que tengan el mismo número de dígitos se proceden a comparar los dígitos de ambos números, desde el más significativo hasta el menos significativo, hasta encontrar una desigualdad o hasta terminar de recorrerlos. Si el ciclo de las instrucciones 15 a 18 termina porque la **i** es mayor que la **m** significa que los números son iguales, por lo tanto retorna 0 (instrucciones 19 a 21). En caso contrario se determina cuál de los dos dígitos diferentes es mayor (instrucciones 22 a 26) y se retorna el dato apropiado.

Resumen

En este módulo hemos visto la herencia en programación orientada a objetos. Adicionalmente tratamos el tema del polimorfismo dinámico con un ejemplo de aplicación.

Ejercicios propuestos

1. Construya un algoritmo para determinar y retornar el residuo (operador módulo) de una división entre dos números enteros.

2. Construya un algoritmo que calcule y retorne el factorial de un número entero bajo esta representación.

3. Construya un algoritmo para calcular y retornar el resultado de la potenciación entre dos números representados como objetos de altaPrecision.

4. Construya un algoritmo que calcule y retorne el máximo común divisor de dos números representados como objetos de altaPrecision.

5. Construya un algoritmo que calcule y retorne el mínimo común múltiplo de dos números representados como objetos de altaPrecision.

Módulo 43

La clase matriz

Introducción

En los módulos 41 y 42 tratamos la clase vector, vimos su definición, sus métodos y la forma de trabajar con ellos. Veremos aquí el tratamiento objetual de matrices.

Objetivo del módulo

1. Representar y manipular matrices como objetos.

Preguntas básicas

1. ¿Cuáles son los datos privados en la definición de una clase matriz?
2. ¿Qué diferencia importante hay entre la manipulación procedimental y la manipulación objetual de matrices?

Contenidos del módulo

43.1 Definición de la clase matriz y sus métodos
43.2 Desarrollo de los algoritmos de los métodos de la clase matriz

«Las matrices representadas como objetos facilitan enormemente el trabajo sobre ellas. El hecho de manejar el arreglo de dos dimensiones y las dimensiones de la matriz como atributos privados evita el permanente paso de parámetros para ejecutar todas las operaciones que con ellas se efectúan. Además, permite recibir objetos de esta clase en los parámetros y crear objetos de esta clase y retornarlos a los programas llamantes sin mucha dificultad».

Vea en el botón **Objetual** del mapa conceptual el video "Matriz como un objeto".

43.1 Definición de la clase matriz y sus métodos

Como vimos anteriormente, una matriz es un objeto matemático que se presenta en una gran variedad de problemas. En general, una matriz consiste de un arreglo con **m** filas y **n** columnas de datos.

mat	1	2	3	4	5	6
1	1	4	8	2	7	– 3
2	4	4	8	6	3	1
3	6	7	5	7	9	8
4	2	4	2	8	1	9
5	6	6	3	6	1	2
6	3	5	5	6	8	4

Figura 43.1

Tradicionalmente, una matriz se almacena en un arreglo de dos dimensiones. Cualquier elemento se puede trabajar designándolo por **mat[i][j],** donde **i** representa la fila y **j** la columna, y su acceso es muy rápido. Para manipular matrices con esta representación definimos una clase matriz de la siguiente forma:

```
1.   clase  matriz
2.        Privado:
3.             entero  filas, columnas
4.             objeto  mat[][]
5.        Público:
6.             matriz(entero  m,  entero  n)              // constructor
7.             entero  numeroFilas()
8.             entero  numeroColumnas()
9.             void  asignaDato(entero  i,  entero  j,  objeto  dato)
10.            objeto  retornaDato(entero  i,  entero  j)
11.            void  muestraMatriz()
12.            matriz  suma(matriz  b)
13.            matriz  multiplica(matriz  b)
14.            matriz  transpuesta()
15.            matriz  inversa()
16.            entero  determinante()
17.  Fin(clase matriz)
```

En esta clase se define el constructor (línea 6) y métodos para conocer el número de filas de la matriz, el número de columnas de la matriz, asignar un dato a una posición específica de la matriz, retornar el dato almacenado en una posición específica, escribir la matriz, sumar matrices, multiplicar matrices, construir la transpuesta de una matriz, construir la inversa de una matriz y calcular el valor del determinante asociado a una matriz cuadrada.

43.2 Desarrollo de los algoritmos de la clase matriz

Comencemos viendo el algoritmo para el constructor.

```
1.   matriz(entero  m,  entero  n)
2.        Variables:  i, j:  numericaEntera
3.        INICIO
```

```
4.              filas = m
5.              columnas = n
6.              objeto  mat = new  objeto[m + 1][n + 1]
7.              PARA  i  DESDE  1  HASTA  m  CON_VARIACION  +1
8.                      PARA  j  DESDE  1  HASTA  n  CON_VARIACION  +1
9.                              mat[i][j]  =  null
10.                     Fin(PARA)
11.             Fin(PARA)
12.         FIN
13.  Fin(matriz)
```

Este algoritmo tiene dos parámetros que son las dimensiones de la matriz a representar. Como en lenguajes de programación como C# y JAVA las matrices definidas con **m** filas y **n** columnas manejan subíndices desde 0 hasta **m – 1** y desde 0 hasta **n – 1**, nosotros, por simplicidad y concordancia con la forma como hemos venido trabajando, definimos la matriz con **m + 1** filas y **n + 1** columnas con el fin de poder manipular los subíndices desde 1 hasta **m** para las filas y desde 1 hasta **n** para las columnas.

(**Atención**: en este texto omitimos la fila 0 y la columna 0. Para propósitos de ciertos problemas que se verán en los siguientes textos se utilizarán esta fila 0 y esta columna 0 con fines especiales que facilitan la programación).

Observe también que se define la matriz **mat** de tipo objeto con el fin de poder almacenar en ella cualquier tipo de dato. Adicionalmente, se inicializa cada celda de la matriz en nulo.

Los algoritmos correspondientes a asignar un dato a una posición específica y a obtener el dato almacenado en una celda son bastante sencillos y se presentan a continuación.

```
1.   void  asignaDato(entero  i,  entero  j,  objeto  dato)
2.              mat[i][j]  =  dato
3.   Fin(asignaDato)
```

```
1.   objeto  retornaDato(entero  i,  entero  j)
2.              retorne(mat[i][j])
3.   Fin(retornaDato)
```

El algoritmo para el método *muestraMatriz()* también es sencillo:

```
1.   void  muestraMatriz()
2.         Variables:  i, j:  numericaEntera
3.         INICIO
4.              PARA  i  DESDE  1  HASTA  filas  CON_VARIACION  +1
5.                   PARA  j  DESDE  1  HASTA  columnas  CON_VARIACION  +1
6.                              ESCRIBA(i, j, retornaDato(i, j))
7.                   Fin(PARA)
8.              Fin(PARA)
9.         FIN
10.  Fin(muestraMatriz)
```

Basta con recorrer el arreglo de dos dimensiones perteneciente al objeto e imprimirlo.

El algoritmo correspondiente al método para sumar matrices es el siguiente.

```
1.    matriz  suma(matriz  b)
2.          Variables:  m, n, p, q, i, j:  numericaEntera
3.                      c:  matriz
4.          INICIO
5.                m = numeroFilas()
6.                n = numeroColumnas()
7.                p = b.numeroFilas()
8.                q = b.numeroColumnas()
9.                SI   m != p   v   n != q
10.                     ESCRIBA(“matrices con diferente tamaño no se pueden sumar”)
11.                     retorne(null)
12.               Fin(SI)
13.               c = new  matriz(m, n)
14.               PARA  i  DESDE  1  HASTA  m  CON_VARIACION  +1
15.                     PARA  j  DESDE  1  HASTA  n  CON_VARIACION  +1
16.                               c.asignaDato(i, j, retornaDato(i, j) + b,retornaDato(i, j))
17.                     Fin(PARA)
18.               Fin(PARA)
19.               retorne(c)
20.         FIN
21.   Fin(suma)
```

Este algoritmo es el mismo que se desarrolló en el numeral 38.2. Sin embargo, es conveniente resaltar las diferencias con respecto a ese algoritmo. Este debe ser invocado por un objeto de la clase matriz, sólo tiene un parámetro que es otro objeto de la clase matriz y retorna otro objeto de la clase matriz, el cual contiene el resultado de la suma de la matriz que invocó el método con la matriz enviada como parámetro. Estas diferencias hacen al algoritmo más sencillo y más legible.

Un algoritmo para el método de multiplicación de matrices es el siguiente:

```
1.    matriz  multiplica(matriz  b)
2.          Variables:  m, n, p, q, i, j, k, s:  numericaEntera
3.                      c:  matriz
4.          INICIO
5.                m = numeroFilas()
6.                n = numeroColumnas()
7.                p = b.numeroFilas()
8.                q = b.numeroColumnas()
9.                if   (n != p)  then
10.                         ESCRIBA(“matrices no se pueden multiplicar”)
11.                         retorne(null)
12.               end(if)
13.               matriz  c = new  matriz(m, q)
14.               PARA  i  DESDE  1  HASTA  m  CON_VARIACION  +1
15.                     PARA  j  DESDE  1  HASTA  q  CON_VARIACION  +1
16.                     s = 0
17.                         PARA  k  DESDE  1  HASTA  n  CON_VARIACION  +1
18.                                   s = s + retornaDato(i, k) * b.retornaDato(k, j)
19.                         Fin(PARA)
20.                     c.asignaDato(i, j, s)
21.                     Fin(PARA)
22.               Fin(PARA)
```

```
23.                retorne(c)
24.          FIN
25.    Fin(multiplica)
```

Este algoritmo es el mismo del numeral 39.1. Las observaciones son las mismas que para el algoritmo de la suma. Es un código más sencillo y legible.

```
1.     matriz  transpuesta()
2.           Variables:  m, n, i, j:  numericaEntera
3.                         c:  matriz
4.           INICIO
5.                 m = numeroFilas()
6.                 n = numeroColumnas()
7.                 c = new  matriz(n, m)
8.                 PARA  i  DESDE  1  HASTA  m  CON_VARIACION  +1
9.                       PARA  j  DESDE  1  HASTA  n  CON_VARIACION  +1
10.                            c.asignaDato(j, i, retornaDato(i, j))
11.                      Fin(PARA)
12.                Fin(PARA)
13.                retorne(c)
14.          FIN
15.    Fin(transpuesta)
```

Como se puede observar los algoritmos para manipular las matrices bajo esta representación son más bien sencillos, considerando que el lector ya tiene cierta experiencia en desarrollo de algoritmos, y la eficiencia de ellos es normal.

Resumen

En este módulo vimos la definición de la clase matriz junto con sus atributos privados, sus métodos públicos y los algoritmos correspondientes a cada uno de ellos.

Ejercicio propuesto

1. Construya una clase matriz con sus respectivos métodos y algoritmos que le permitan resolver los ejercicios propuestos en el módulo 39 (excepto el 1, que corresponde a un seguimiento).

Bibliografía

Joyanes Aguilar, L. (2003), *Fundamentos de programación*, Madrid, McGraw-Hill.

Oviedo, E. (2002), *Lógica de programación,* Bogotá, Editorial ECOE.

Tremblay, J. P. (1982), *Introducción a la ciencia de los computadores, un enfoque algorítmico,* México, McGraw-Hill.

Cibergrafía

Algoritmia básica: http://www.mis-algoritmos.com/

Algoritmos: http://www.geocities.com/David_ees/Algoritmia/indice.htm

El poder de lo simple: http://riosur.net/

Ude@

Para ser, saber y saber hacer

Educación virtual